ARTIFICIAL INTELLIGENCE PART 3 – PROBLEM SOLVING - SEARCHING

Njeri Ireri

July – October 2021

We Shall Discuss

- Introduction to problem solving
- Problem Solving Techniques
- Search as a problem solving technique
- Problem Definition
- Search Terminology
- Evaluating a search

Workers are always Searching

Brief info on Ants, http://www.winnipeg-bugline.com/ants.html,

http://www.boston.com/globe/magazine/2002/0623/frontiers.htm

Problem Solving Techniques in A.I

- *Broad* Approaches
  - using search techniques – uninformed, informed…
    - e.g. in Games
  - modeling
    - using Knowledge Base Systems (KBS)
    - using Machine Learning techniques e.g. Artificial Neural Networks, Decision Trees, Case-base reasoning, Genetic algorithms, ..

Searching as a Problem Solving Technique

- **Searching** is the process of looking for the solution of a problem through a set of possibilities (state space)
- **Search** conditions include:
  - Current state – where one is;
  - Goal state – the solution reached; check whether it has been reached;
  - Cost of obtaining the solution
- The **solution** is a path from the current state to the goal state

Searching as a Problem Solving Technique

Process of Searching

- Searching proceeds as follows:
  - Check the current state;
  - Execute allowable actions to move to the next state;
  - Check if the new state is the solution state; if it is not, then the new state becomes the current state and the process is repeated until a solution is found or the state space is exhausted

Search Problem

- The **search problem** consists of finding a **solution plan,** which is a path from the current state to the goal state
- **Representing search problems**
  - A search problem is represented using a directed graph (tree)
  - The states are represented as nodes while the allowed steps or actions are represented as arcs (branches)
- **A search problem is defined by specifying:**
  - State space;
  - Start node;
  - Goal condition, and a test to check whether the goal condition is met;
  - Rules giving how to change states
  - Path cost

Problem Definition - Example, 8 puzzle

5	4	
6	1	8
7	3	2

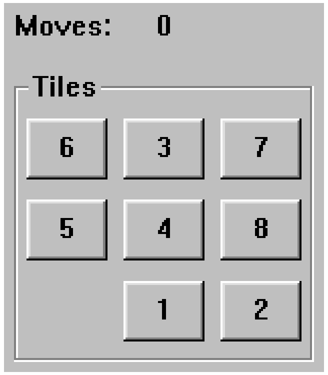
Initial State

1	4	7
2	5	8
3	6	

Goal State

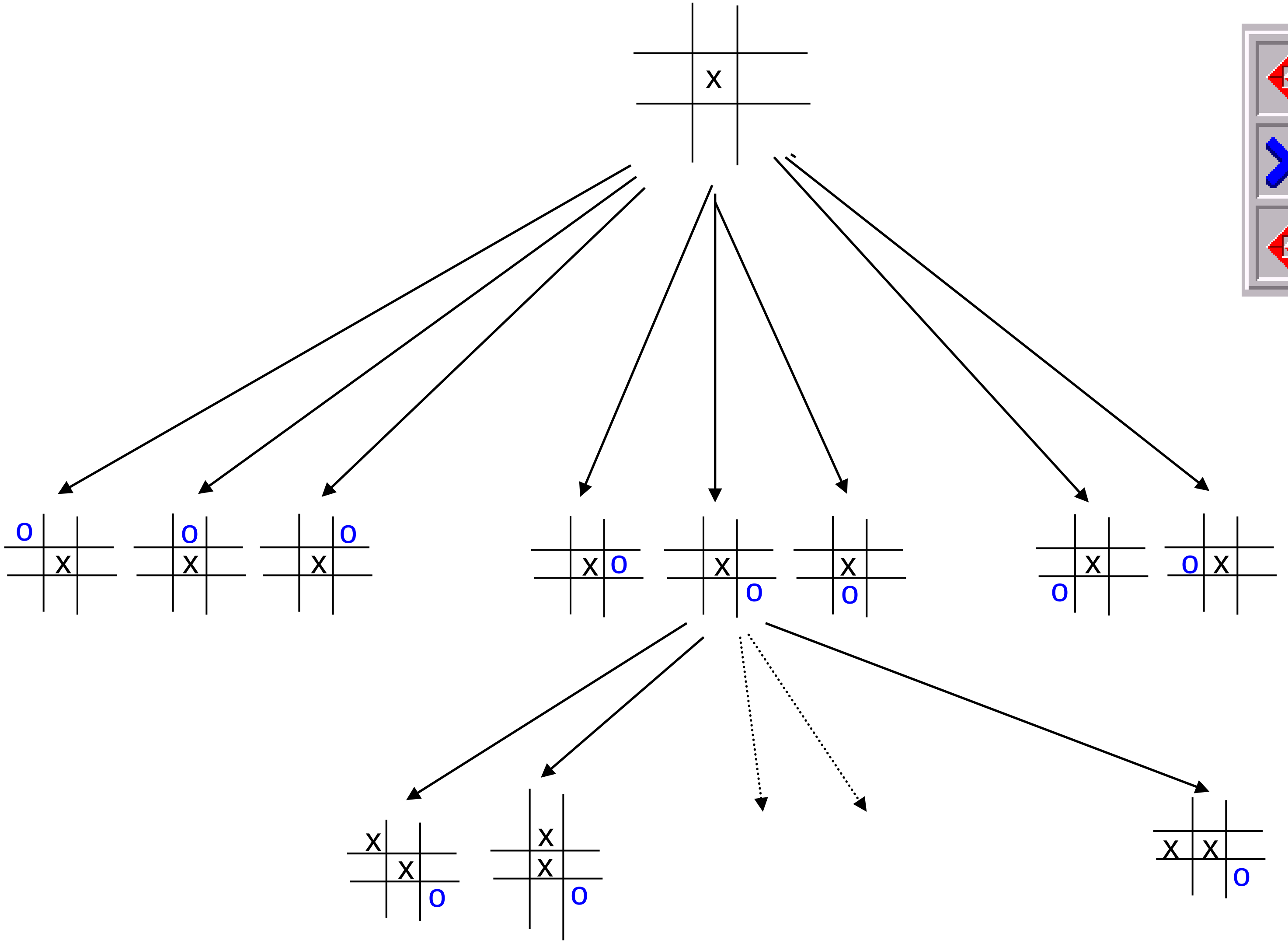
Problem Definition - Example, 8 puzzle

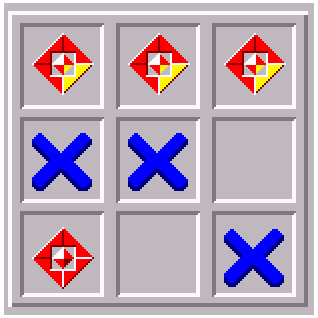
- States
  - A description of each of the eight tiles in each location that it can occupy. It is also useful to include the blank



- Operators/Action
  - The blank moves left, right, up or down
- Goal Test
  - The current state matches a certain state (e.g. one of the ones shown on previous slide)
- Path Cost
  - Each move of the blank costs 1

Problem Definition - Example, tic-tac-toe





Exercise

4. (a) Playing the 8 Puzzle game, draw a search tree to level three for the initial game state given in figure 1

   (b) How many moves would you require to complete the game given in figure 1

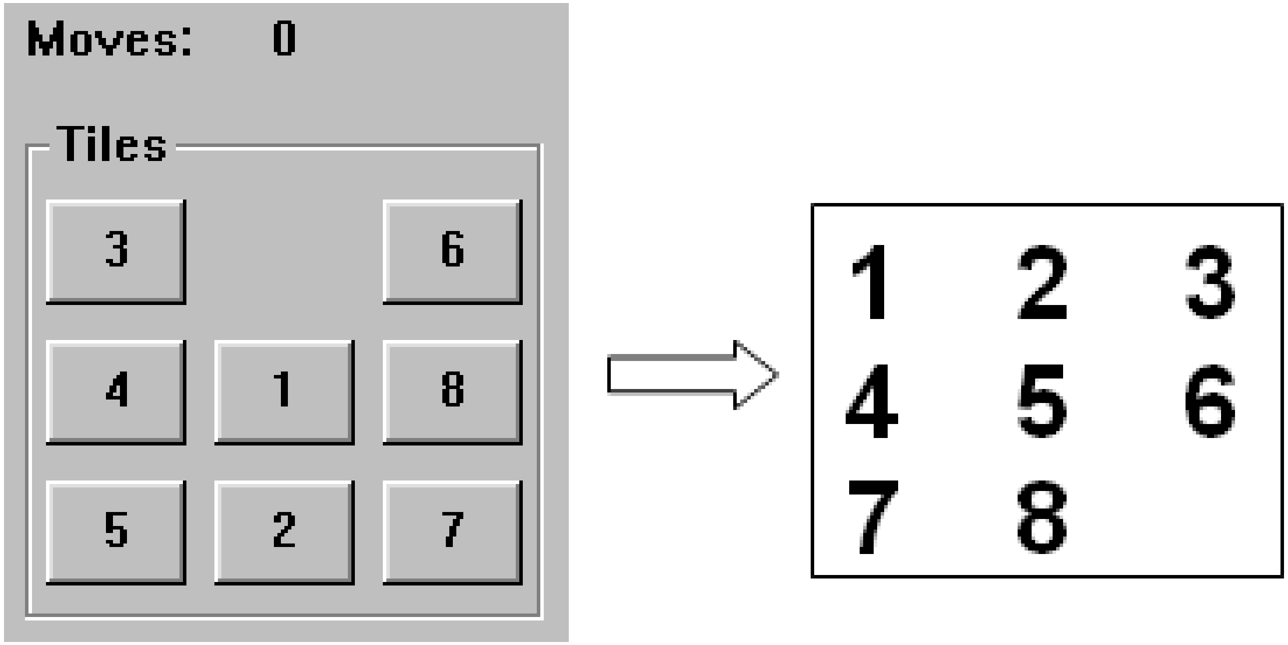
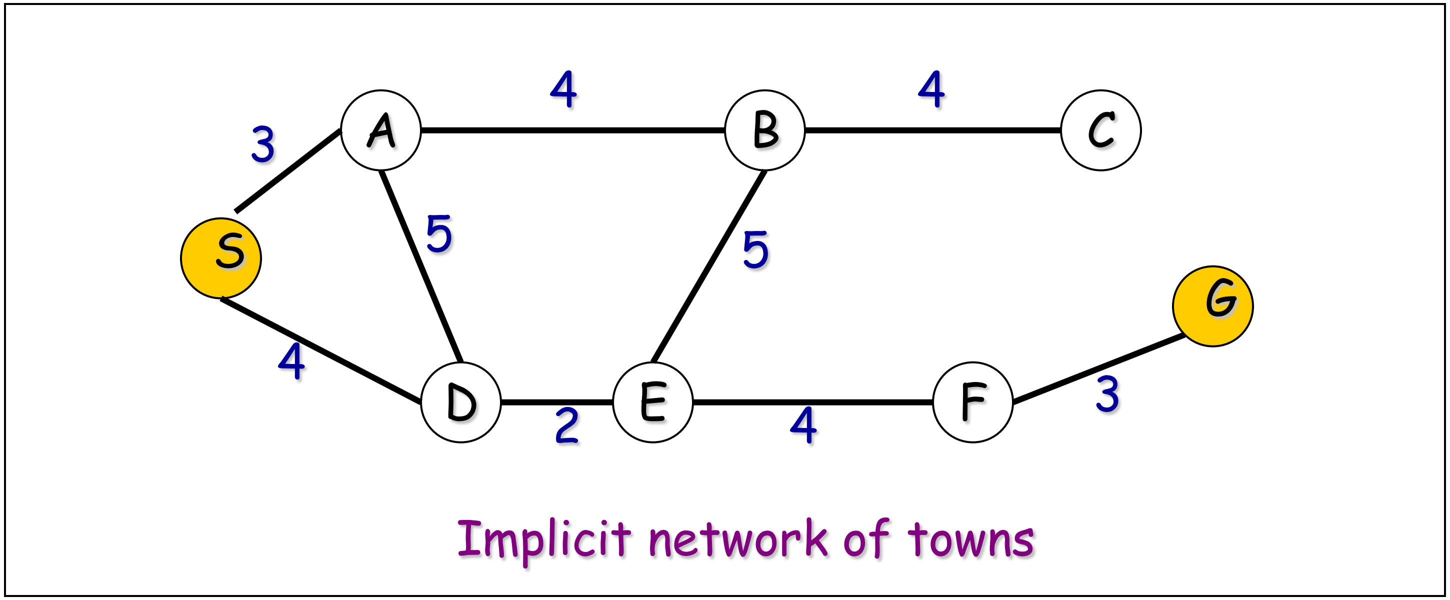


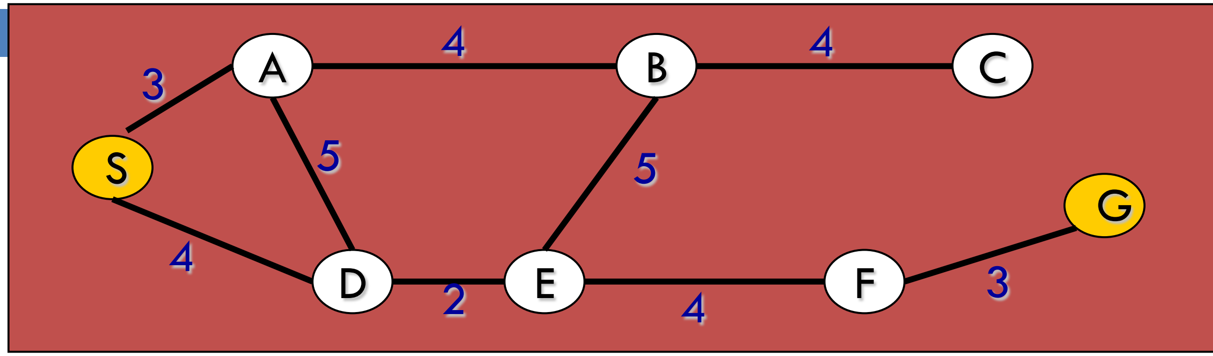
Figure 1. 8 Puzzle (Left-Start State and Right-Goal State)

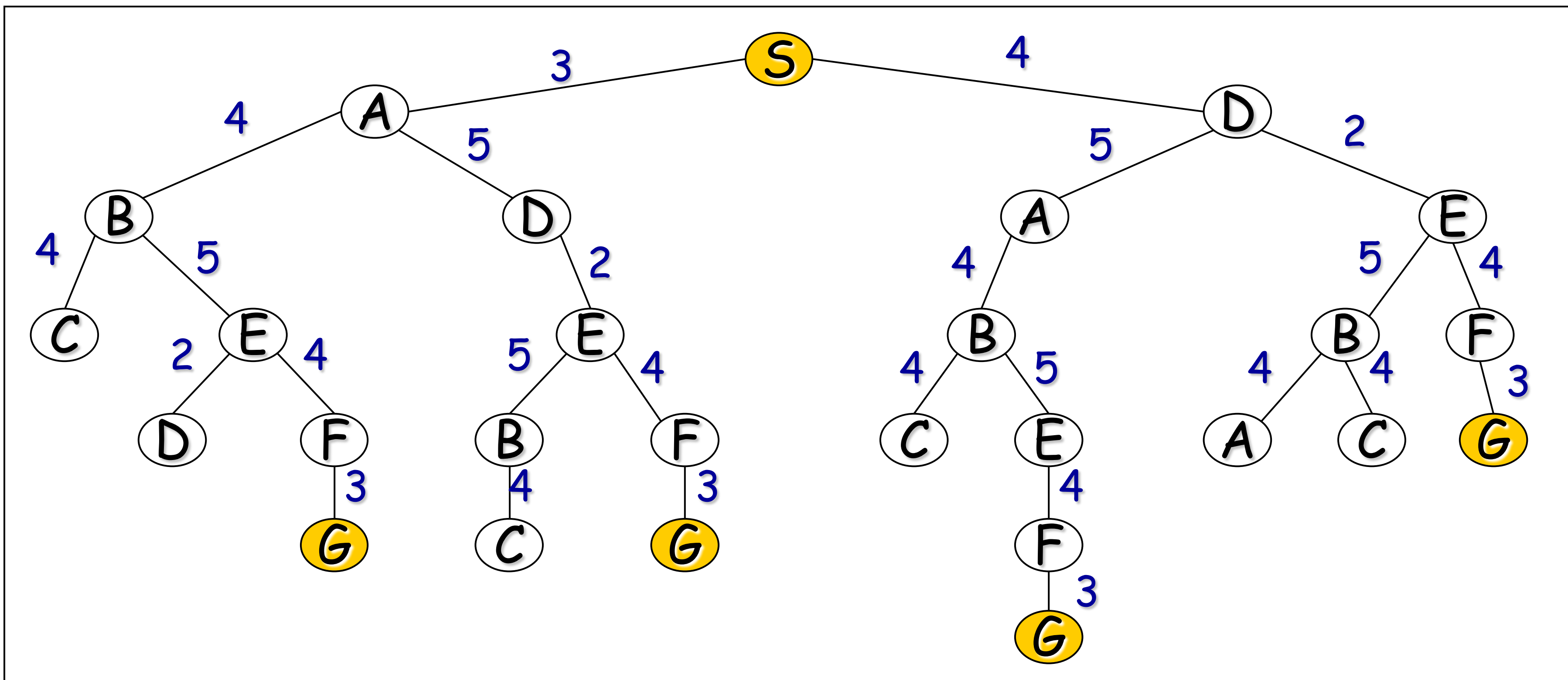
Tree/Path Example:



- Two possible tasks:
  - 1. FIND a (the) path. = computational cost
  - 2. TRAVERSE the path. = travel cost
- 2. relates to finding optimal paths

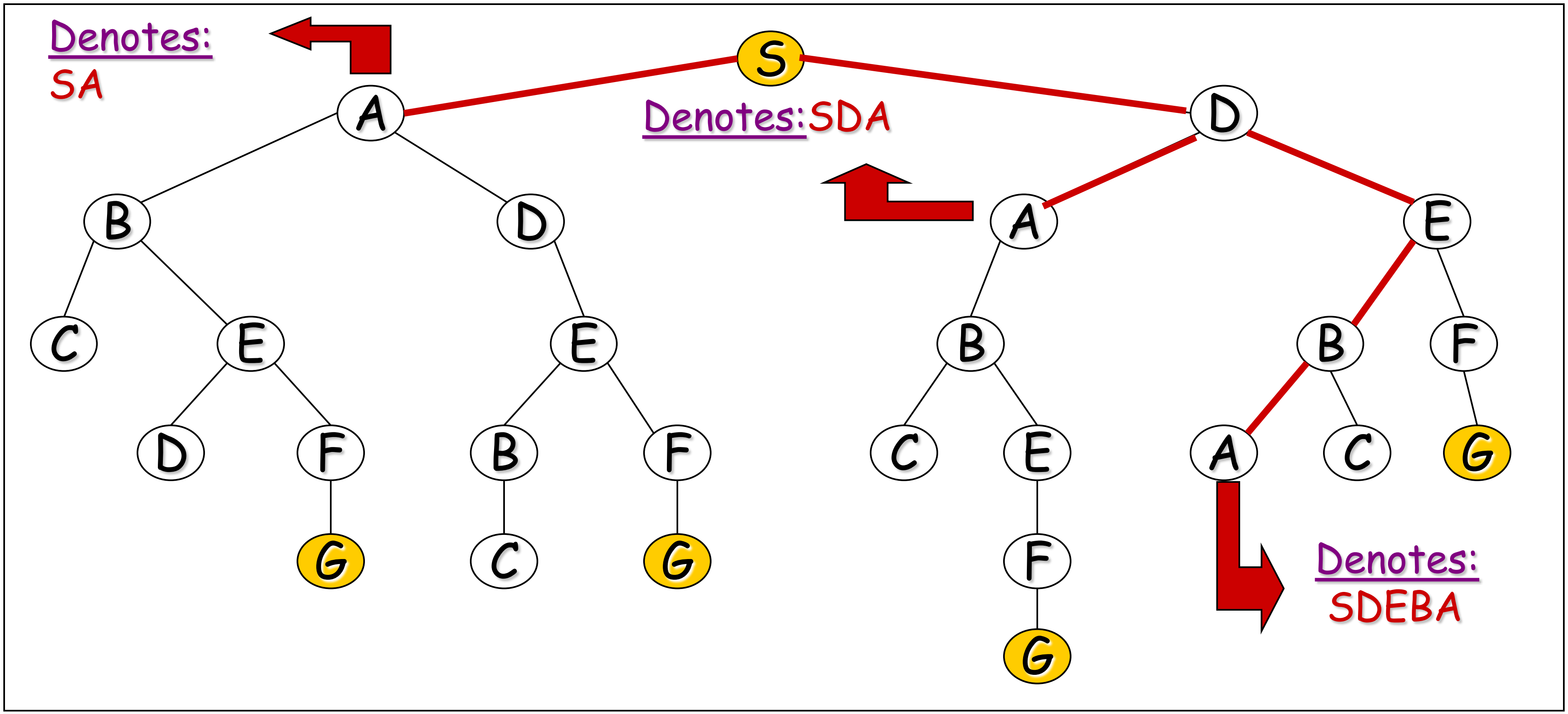
The associated loop-free tree of partial paths





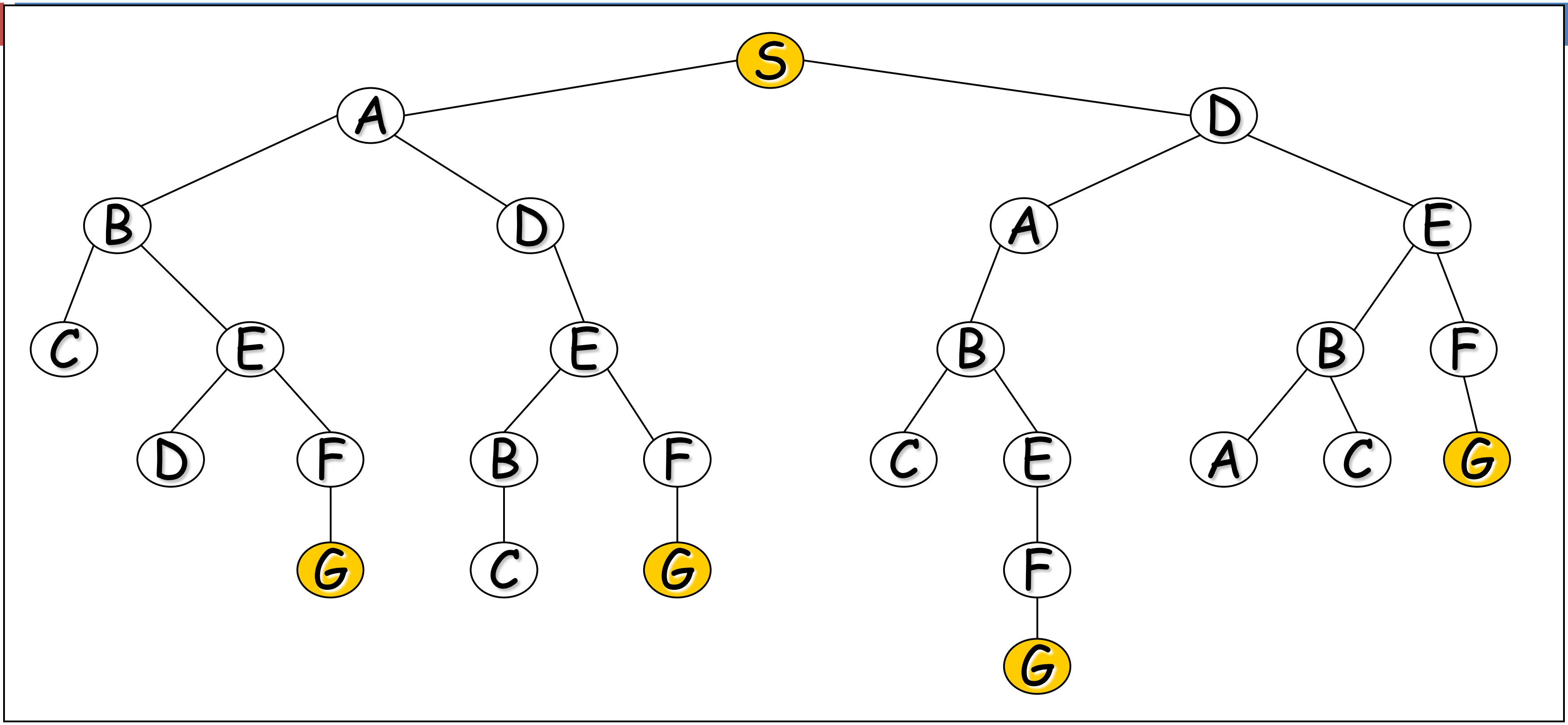
Paths:

- We are not interested in optimal paths here, so we can drop the costs



Note: Nodes do not denote themselves, but denote the partial path from the root to themselves!!

Terminology:



- Node, link (or edge), branch, arc
- Parent, child, ancestor, descendant
- Root node, goal node
- Expand / Open node / Closed node / Branching factor

Using a Tree – The Obvious Solution?

- **But**
  - It can be wasteful on space
  - It can be difficult to implement, particularly if there are varying number of children (as in tic-tac-toe)
  - It is not always obvious which node to expand next
    - We may have to search the tree looking for the best **leaf node** (sometimes called the **fringe** or **frontier** nodes). This can obviously be computationally expensive

How Good is a Solution?

- Does our search method actually find a solution?
- Is it a good solution?
  - Path Cost
  - Search Cost (Time and Memory)
- Does it find the optimal solution?
  - But what is optimal?

Evaluating a Search

- Completeness
  - Is the strategy guaranteed to find a solution?
- Time Complexity
  - How long does it take to find a solution?
- Space Complexity
  - How much memory does it take to perform the search?
- Optimality
  - Does the strategy find the optimal solution where there are several solutions?

Search Trees

- Some issues:
  - Search trees grow very quickly
  - The size of the search tree is governed by the branching factor
  - Even this simple game tic-tac-toe has a complete search tree of 984,410 potential nodes
  - The search tree for chess has a branching factor of about 35

Exercise

1. How are problems solved in artificial intelligence?
2. What is searching?
3. (a) What are the things that could specify a search problem?

   (b) Supposing you had a robot that is supposed to maneuver it self on a factory floor cluttered with numerous machines and boxes containing both raw and finished materials from the back to the front of the factory. What would be specified for the case in (a)

Search Techniques

- Uninformed Search – Blind Search
- Informed Search – Heuristic Search

Part 3.1 – UNINFORMED (BLIND) SEARCH

We Shall Discuss

- What is Uninformed (Blind) Search?
- Uninformed Search Methods
  - Depth-first search
  - Breadth-first search
  - Non-deterministic search
  - Iterative deepening search
  - Bi-directional search

Uninformed Search?

- Simply searches the state space (or NET)
- Can only distinguish between goal state and non-goal state
- Sometimes called Blind search as it has no information or knowledge about its domain

Uninformed Search Characteristics

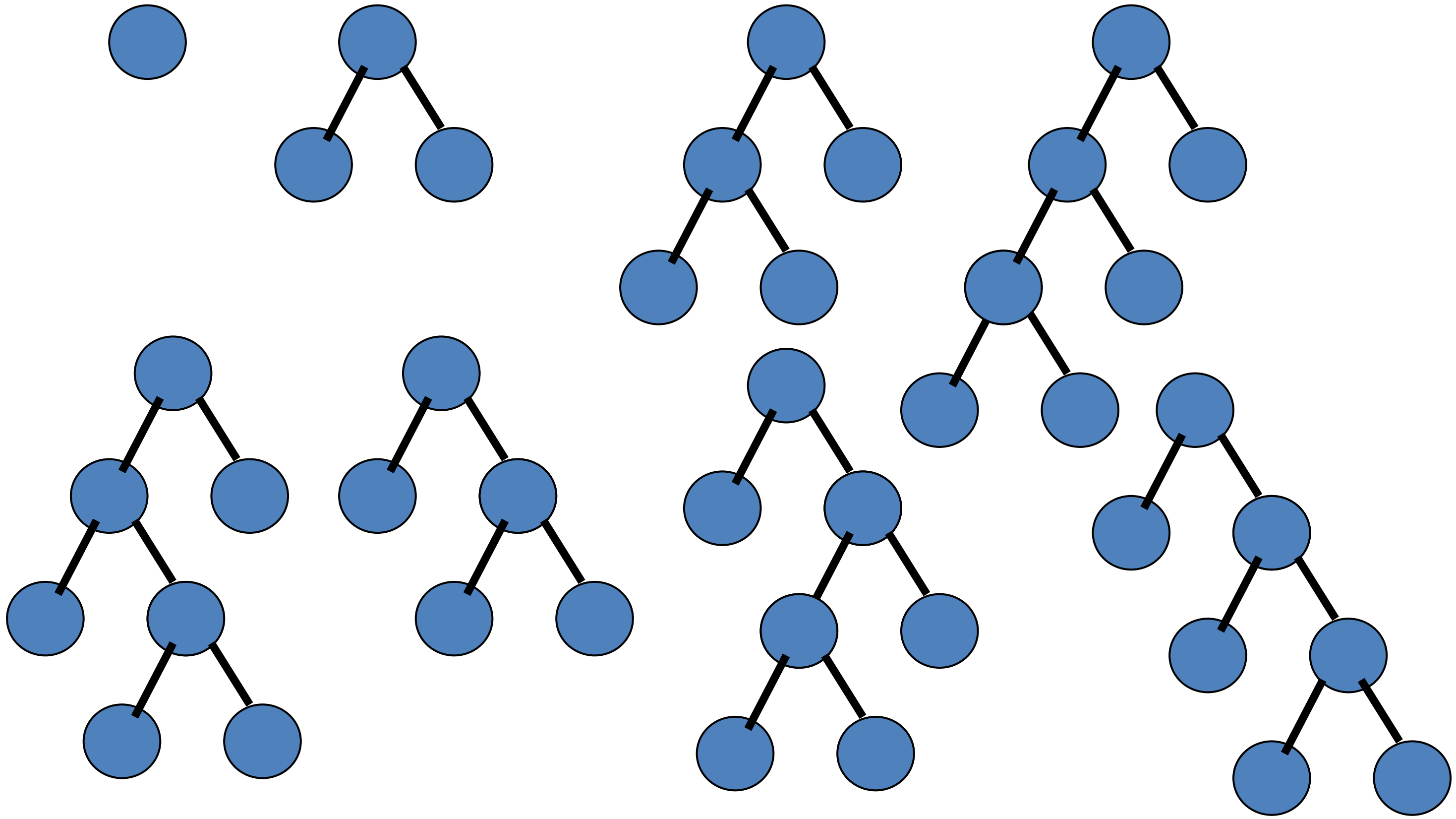
- Blind Searches have no preference as to which state (node) that is expanded next
- The different types of blind searches are characterised by the order in which they expand the nodes
  - This can have a dramatic effect on how well the search performs when measured against the four criteria we defined in an earlier lecture
  - Search evaluation criteria - Completeness, Time Complexity, Space Complexity, Optimality (of given solution when there are several solutions to choose from)

Uninformed (Blind) Search Methods

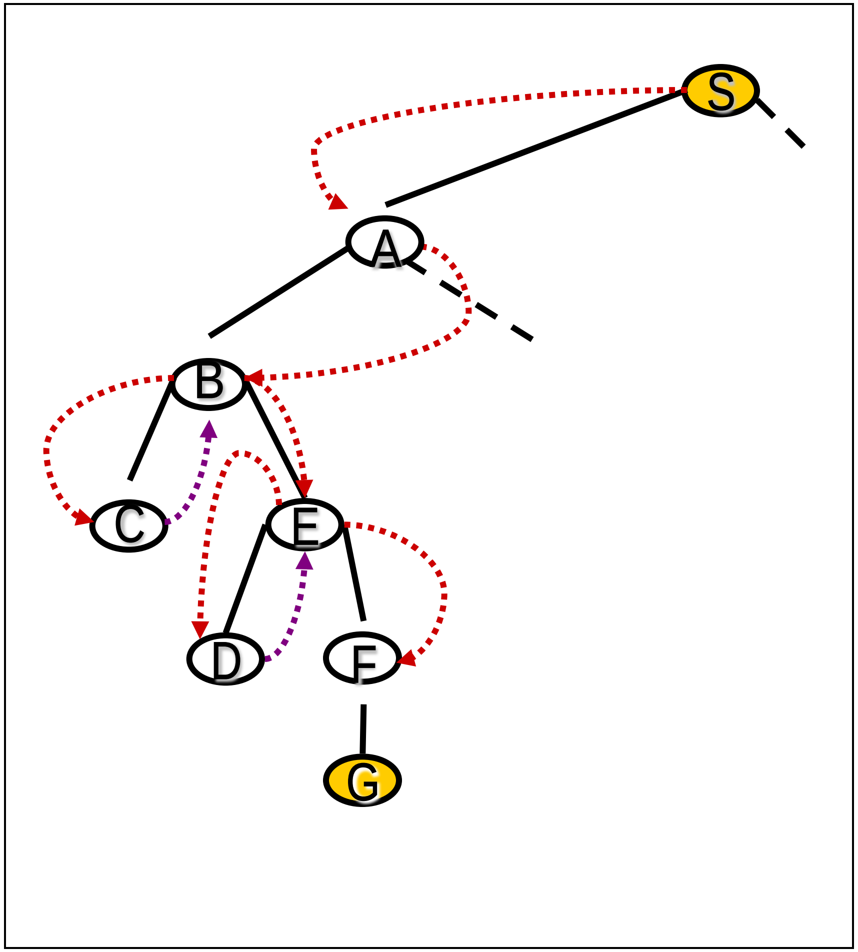
- Methods that do not use any specific knowledge about the problem
- These are:
  - Depth-first search
  - Breadth-first search
  - Non deterministic search
  - Iterative deepening search
  - Bi-directional search

1. Depth-first Search

- Expand the tree as deep as possible, returning to upper levels when needed



Depth-First Search = Chronological backtracking



- Select a child
  - convention: left-to-right
- Repeatedly go to next child, as long as possible
- Return to left-over alternatives (higher-up) only when needed

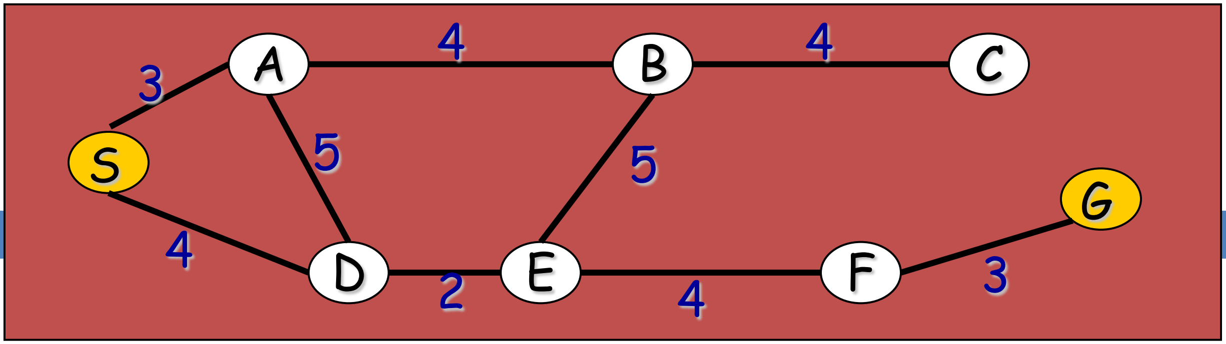
Depth-First Algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
AND goal is not reached

DO remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
add the new paths to front of QUEUE;

3. IF goal reached
THEN success;
ELSE failure;



1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty AND goal is not reached

   DO
   - remove the first path from the QUEUE;
   - create new paths (to all children);
   - reject the new paths with loops;
   - add the new paths to front of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

Trace of Depth-First for running example:

(S)	S removed, (SA,SD) computed and added
(SA, SD)	SA removed, (SAB,SAD,SAS) computed, SAB,SAD) added
(SAB,SAD,SD)	SAB removed, (SABA,SABC,SABE) computed, (SABC,SABE) added
(SABC,SABE,SAD,SD)	SABC removed, (SABCB) computed, nothing added
(SABE,SAD,SD)	SABE removed, (SABEB,SABED,SABEF) computed, (SABED,SABEF)added
(SABED,SABEF,SAD,SD)	SABED removed, (SABEDS,SABEDA.SABEDE) computed, nothing added
(SABEF,SAD,SD)	SABEF removed, (SABEFE,SABEFG) computed, (SABEFG) added
(SABEFG,SAD,SD)	goal is reached: reports success

Evaluation Criteria:

- **Completeness**
  - Does the algorithm always find a path?
    - (for every state space such that a path exits)
- **Speed** (worst time complexity) :
  - What is the highest number of nodes that may need to be created?
- **Memory** (worst space complexity) :
  - What is the largest amount of nodes that may need to be stored?
- Expressed in terms of:
    - d = depth of the tree
    - b = (average) branching factor of the tree
    - m = depth of the shallowest solution

Note: approximations !!

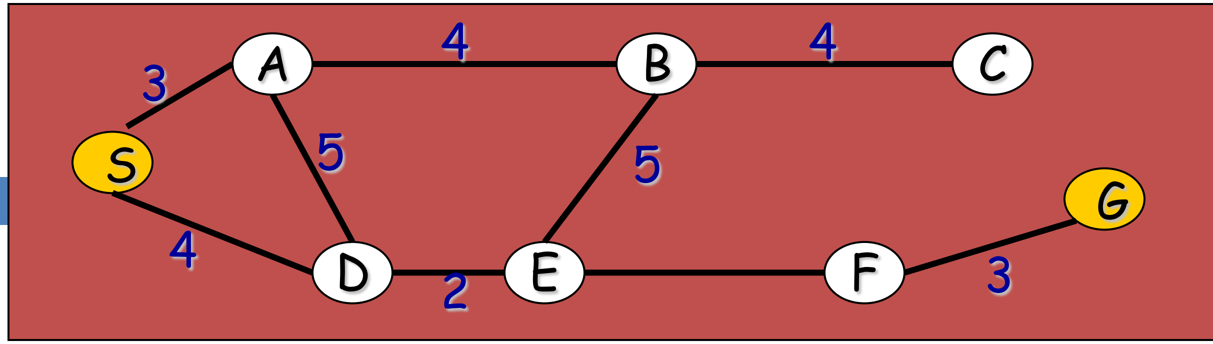
- In our complexity analysis, we do not take the built-in **loop-detection** into account
- The results only 'formally' apply to the variants of our algorithms **WITHOUT** loop-checks
- Studying the effect of the loop-checking on the complexity is hard:
  - The overhead of the checking MAY or MAY NOT be compensated by the reduction of the size of the tree
- Also: our analysis **DOES NOT** take the length (space) of representing paths into account !!

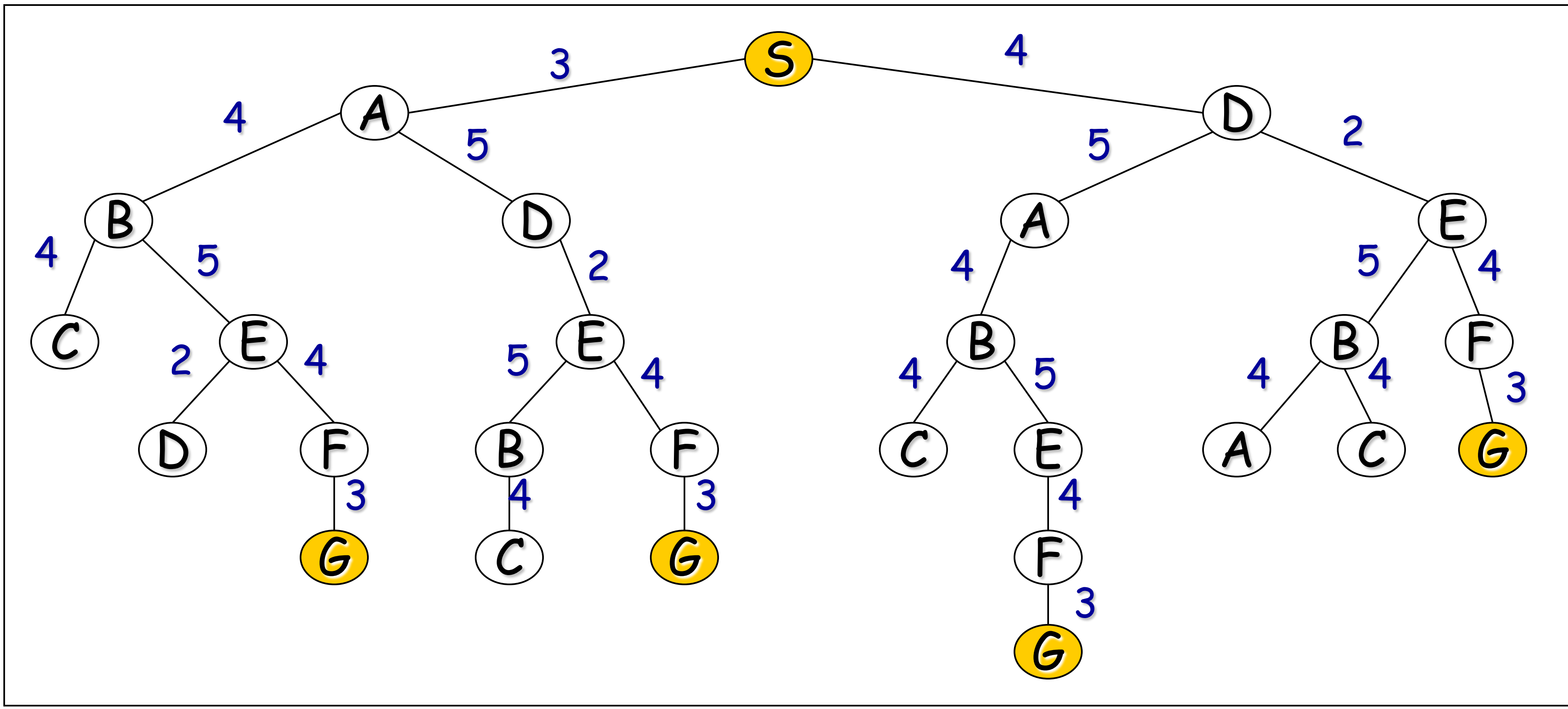
Completeness (depth-first)

- Complete for FINITE (implicit) NETS
  - (= State space with finitely many nodes)

- **IMPORTANT:**
  - This is due to integration of LOOP-checking in this version of Depth-First (and in all other algorithms that will follow) !
    - IF we do not remove paths with loops, then Depth-First is not complete (may get trapped in loops of a finite State space)

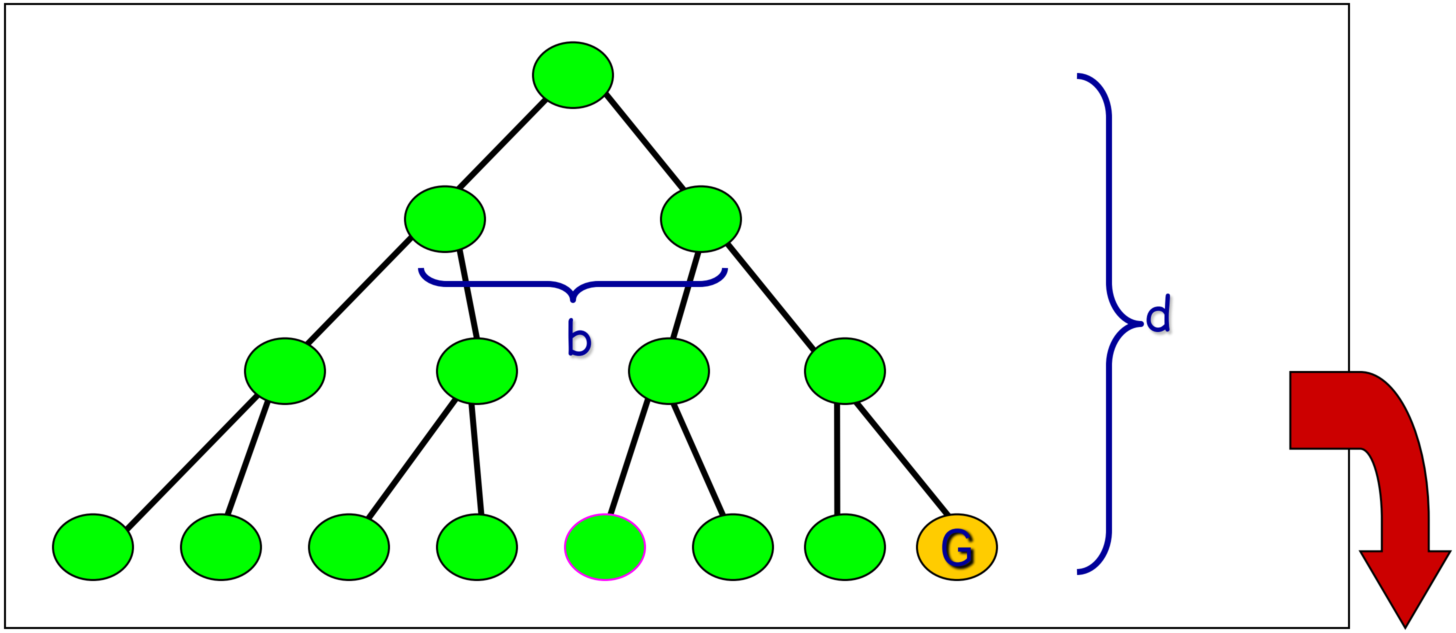
- **Note:** does NOT find the shortest path


S
A
B
C
D
E
F
G
3
4
4
5
5
4
2
3


S
A
D
B
C
E
F
G
3
4
5
2

Speed (depth-first)

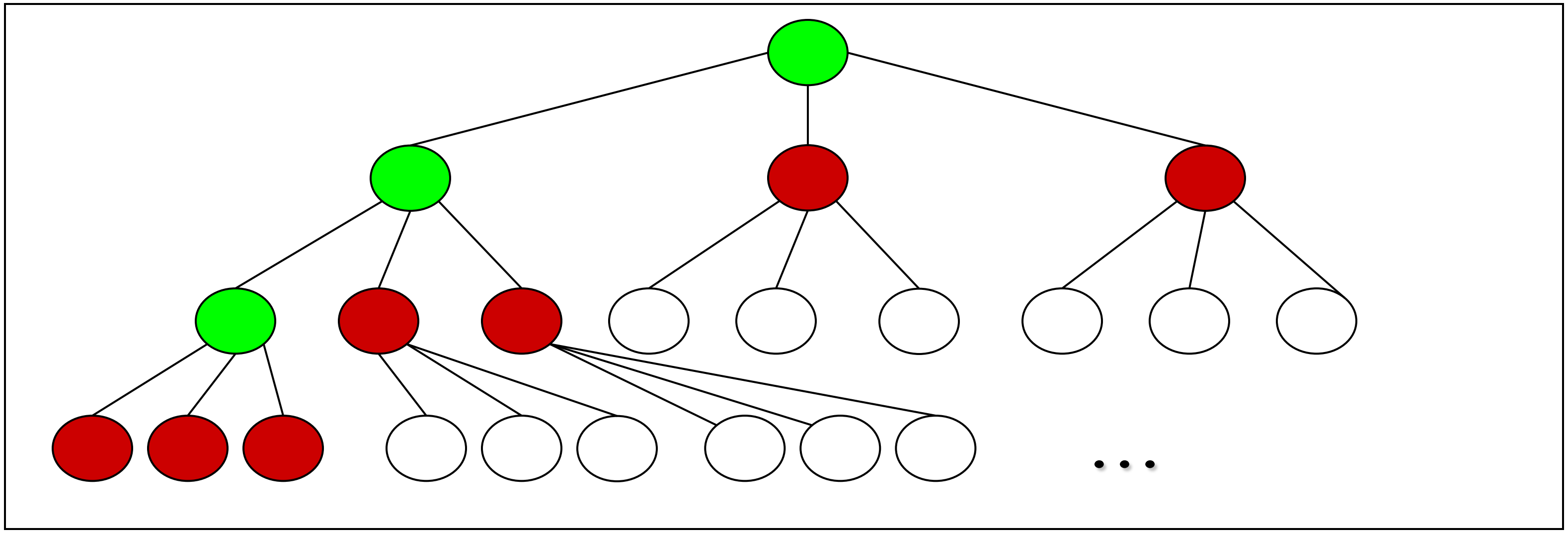
- In the worst case:
  - the (only) goal node may be on the right-most branch,

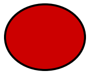


- Time complexity == $b^d + b^{d-1} + \ldots + 1 = \dfrac{b^{d+1} - 1}{b - 1}$
- Thus: $O(b^d)$

Memory (depth-first)

- Largest number of nodes in QUEUE is reached in bottom left-most node
- Example: d = 3, b = 3 :



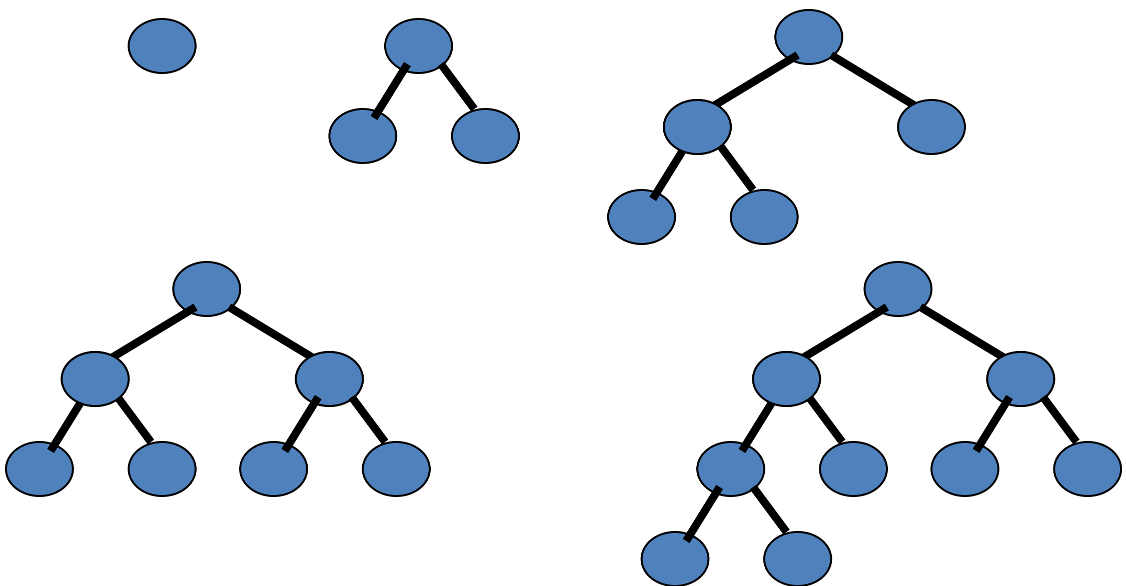
  - QUEUE contains all  nodes. Thus: 7.
  - In General: ((b-1) * d) + 1
  - Order: O(d*b)

2. Breadth-First Search

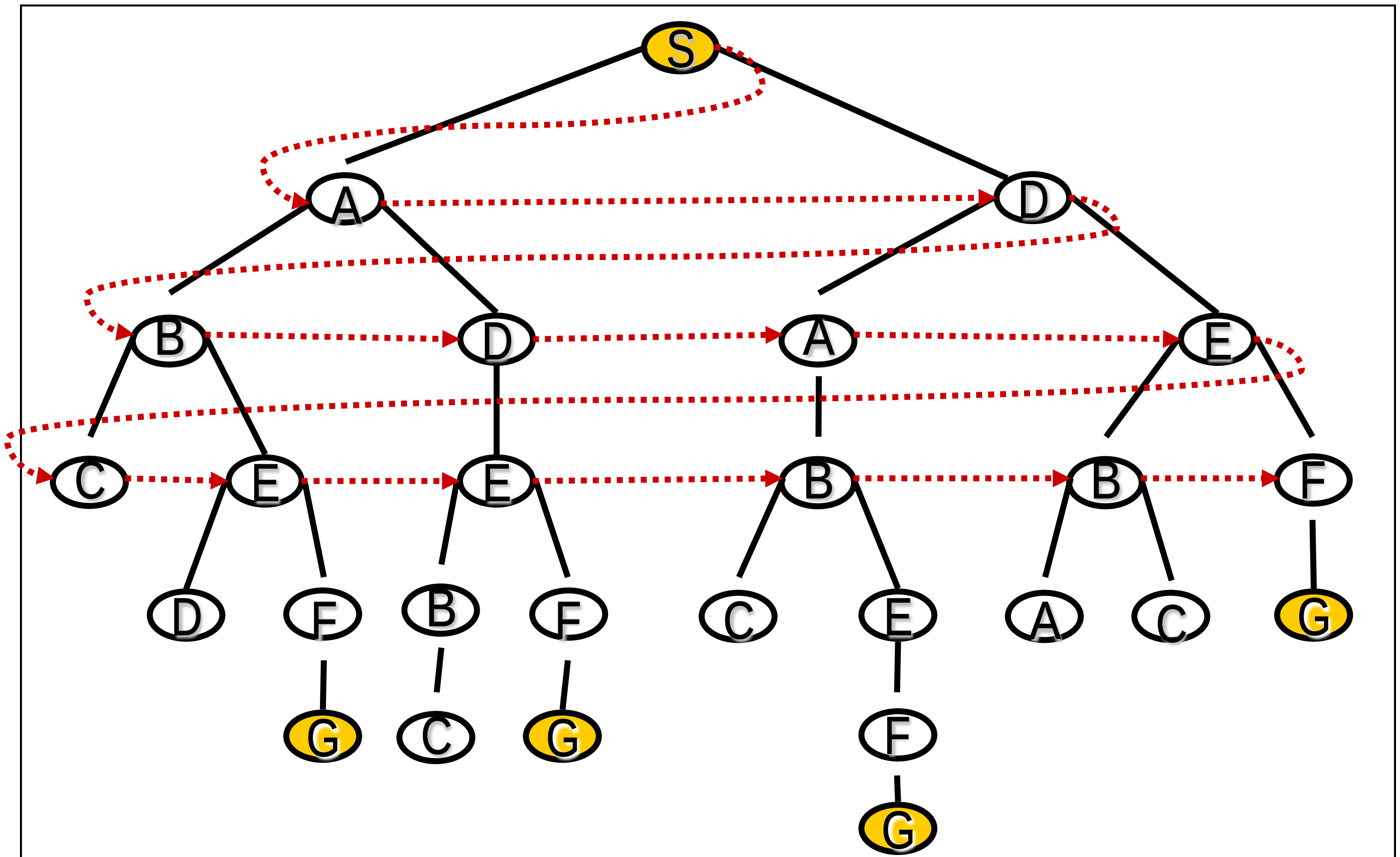
- Expand the tree layer by layer, progressing in depth.
- In other words,
  - Expand root node first
  - Expand all nodes at level 1 before expanding level 2

  OR

  - Expand all nodes at level d before expanding nodes at level d+1



Breadth-First Search:



- Move downwards, level by level, until goal is reached

Breadth-First Algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE { QUEUE is not empty
   AND goal is not reached

   DO { remove the first path from the QUEUE;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths to back of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

ONLY DIFFERENCE !

Trace of breadth-first for running example:

- (S) S removed, (SA,SD) computed and added
- (SA, SD) SA removed, (SAB,SAD,SAS) computed, (SAB,SAD) added
- (SD,SAB,SAD) SD removed, (SDA,SDE,SDS) computed, (SDA,SDE) added
- (SAB,SAD,SDA,SDE) SAB removed, (SABA,SABE,SABC) computed, (SABE,SABC) added
- (SAD,SDA,SDE,SABC,SABE) SAD removed, (SADS,SADA, SADE) computed, (SADE) added
- etc, until QUEUE contains:
- (SABED,SABEF,SADEB,SADEF,SDABC,SDABE,SDEBA,SDEBC, SDEFG) goal is reached: reports success

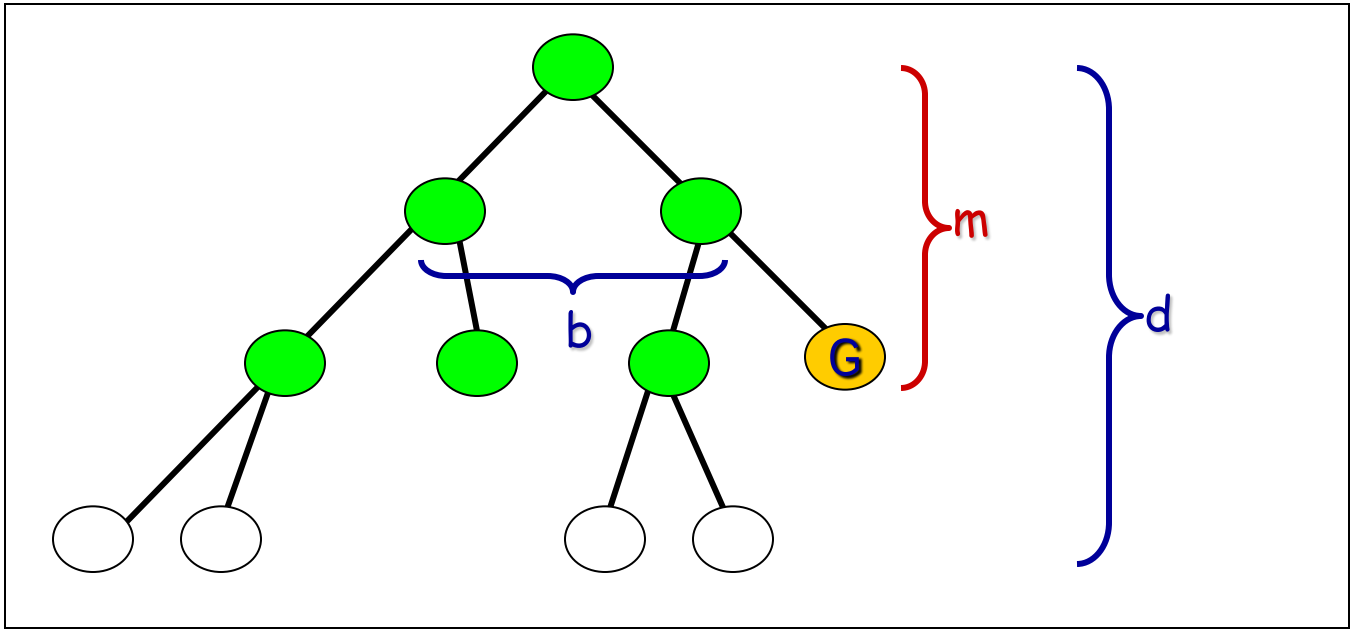
Completeness (breadth-first)

- **Complete**
  - even for infinite implicit NETS !
  - Would even remain complete without our loop-checking

- **Note:** ALWAYS finds the shortest path

Speed (breadth-first)

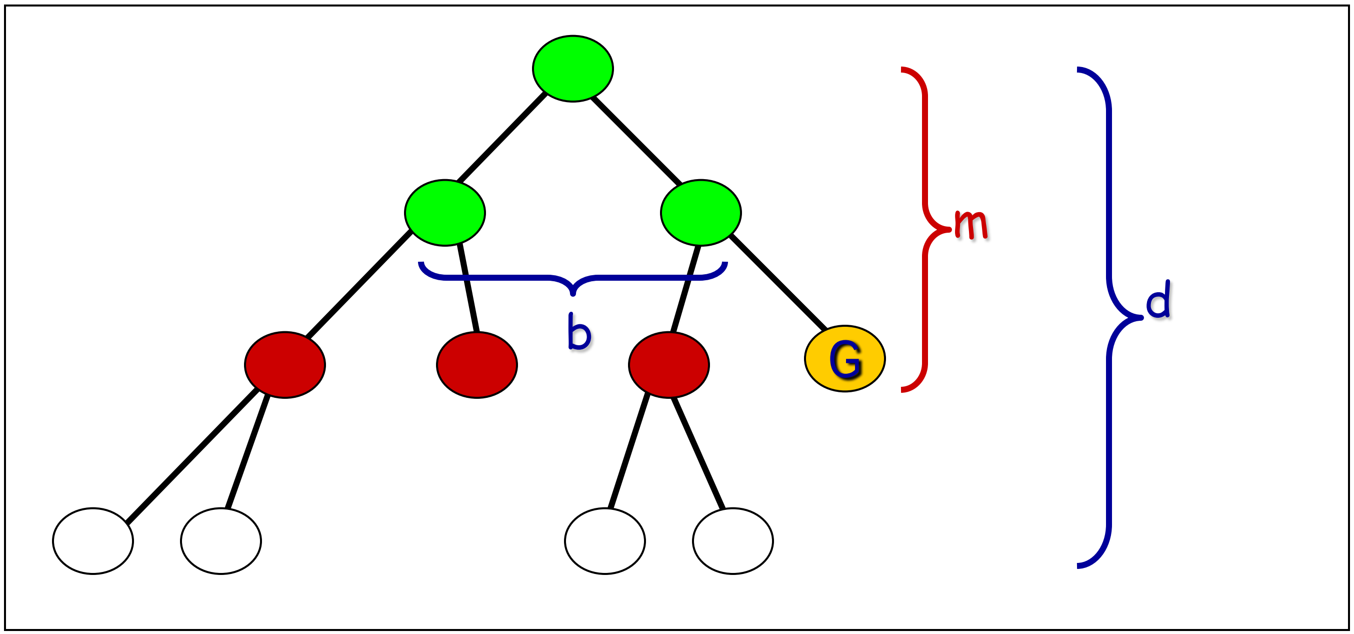
- If a goal node is found on depth m of the tree, all nodes up till that depth are created

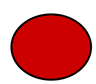


- Thus: $O(b^m)$
- **Note:** depth-first would also visit deeper nodes

Memory (breadth-first)

- Largest number of nodes in QUEUE is reached on the level m of the goal node



  - QUEUE contains all  and  nodes. (Thus: 4)
  - In General: b^m
  - This usually is <u>MUCH</u> worse than depth-first !!

Exponential Growth (breadth-first)

Depth	Nodes	Time		Memory	
0	1	1	millisecond	100	kbytes
2	111	0.1	second	11	kilobytes
4	11,111	11	seconds	1	megabyte
6	10^6	18	minutes	111	megabytes
8	10^8	31	hours	11	gigabytes
10	10^{10}	128	days	1	terabyte
12	10^{12}	35	years	111	terabytes
14	10^{14}	3500	years	11,111	terabytes

- Time and memory requirements for breadth-first search, assuming a branching factor of 10, 100 bytes per node and searching 1000 nodes/second

Exponential Growth - Breadth-First Observations

- Space is more of a factor to breadth first search than time

- Time is still an issue. Who has 35 years to wait for an answer to a level 12 problem (or even 128 days to a level 10 problem)

- It could be argued that as technology gets faster then exponential growth will not be a problem. But even if technology is 100 times faster we would still have to wait 35 years for a level 14 problem and what if we hit a level 15 problem!

Practical Evaluation:

- **1.Depth-first search:**
  - IF the search space contains very deep branches without solution, THEN Depth-first may waste much time in them
- **2. Breadth-first search:**
  - Is VERY demanding on memory !
- **Solutions ??**
  - Non-deterministic search
  - Iterative deepening

3. Non-deterministic Search

- A Non-deterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm
- There are several ways an algorithm may behave differently from run to run

Non-deterministic Search

1. QUEUE <-- path only containing the root;

2. WHILE { QUEUE is not empty
AND goal is not reached

DO { remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
add the new paths in random places in QUEUE;

3. IF goal reached
THEN success;
ELSE failure;

4. Iterative Deepening Search

- Also referred to as **Iterative Deepening Depth-First Search**
- Restrict a depth-first search to a fixed depth
  - a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found
- If no path is found, increase the depth and restart the search

Depth-limited Search

1. DEPTH <-- <some natural number>
   QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty AND goal is not reached

   DO
   - remove the first path from the QUEUE;
   - IF path has length smaller than DEPTH
     - create new paths (to all children);
   - reject the new paths with loops;
   - add the new paths to front of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

Iterative Deepening Algorithm:

1. DEPTH <-- 1

2. WHILE goal is not reached

   DO $\begin{cases}\text{perform Depth-limited search;}\\ \text{increase DEPTH by 1;}\end{cases}$

Iterative Deepening: the best 'blind' search

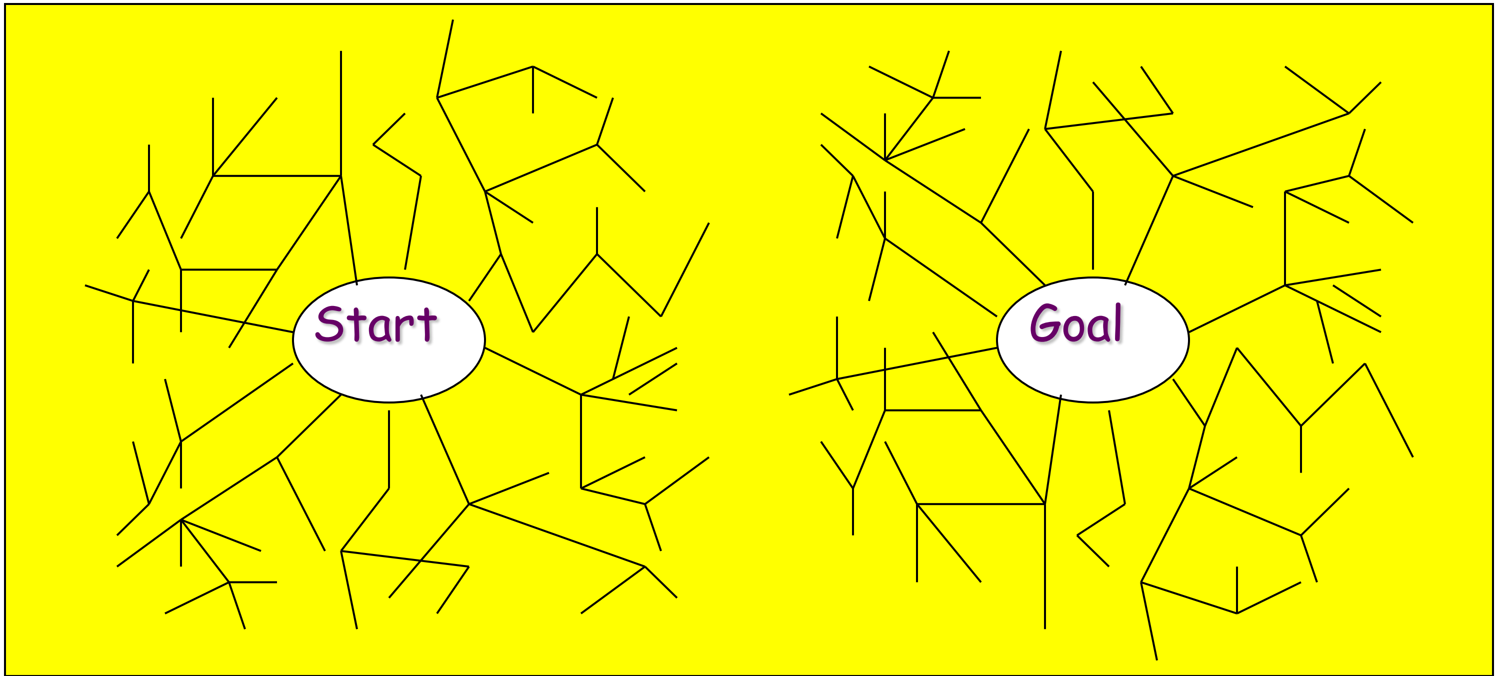
- Complete: yes - even finds the shortest path (like breadth first)
- Memory: $b*m$ (combines advantages of depth- and breadth-first)
- Speed:
  - If the path is found for Depth = m, then how much time was wasted constructing the smaller trees??
- $b^{m-1} + b^{m-2} + \ldots + 1 = \frac{b^m - 1}{b - 1} = O(b^{m-1})$
- While the work spent at DEPTH = m itself is $O(b^m)$

In general: VERY good trade-off

5. Bi-directional Search

- Compute the tree from the start node and from a goal node, until these meet
- IF you are able to EXPLICITLY describe the GOAL state, AND

you have BOTH rules for FORWARD reasoning AND BACKWARD reasoning:



Bi-directional Algorithm:

1. QUEUE1 <-- path only containing the root;
   QUEUE2 <-- path only containing the goal;

2. WHILE both QUEUEi are not empty
   AND QUEUE1 and QUEUE2 do NOT share a state

   DO { remove their first paths;
   create their new paths (to all children);
   reject their new paths with loops;
   add their new paths to back;

3. IF QUEUE1 and QUEUE2 share a state
   THEN success;
   ELSE failure;

Properties (Bi-directional):

- Complete: Yes.
- Speed: If the test on common state can be done in constant time (hashing):
  - $2 * O(b^{m/2}) = O(b^{m/2})$
- Memory: similarly: $O(b^{m/2})$

Part 3.2 – INFORMED (HEURISTIC) SEARCH

We Shall Discuss

- Concept
- Informed search methods

Search with Domain Knowledge added

- Uninformed (blind) searches are normally very inefficient
- Adding domain knowledge can improve the search process
- Concept of Informed (Heuristic) Search
  - Heuristic (informed) search → explore the node that is most "likely" to be the nearest to a goal state
  - There is no guarantee that the heuristic provided most "likely" node will get you closer to a goal state than any other

Knowledge/info

Examples:

- An office in a city building
  - Find me in Office door number N311 e.g. KICC, NSSF, LU Main Campus, Comp Lab building?
    - Wing, floor, left/right
- *Some streets in Nairobi are named in alphabetical order*

- Visit the doctor
  - Symptoms: fever, nausea, headache, …
    - **Leading questions:** how long?, traveled?, … (Malaria, typhoid, meningitis, flu,..)
    - x Blood test, y test,...

Heuristic Searches

- Characteristics
  - Has some **domain knowledge**
  - Usually **more efficient** than blind searches
  - Also **called informed search**
  - Heuristic searches **work by deciding which is the next best node to expand** (there is no guarantee that it is the best node)
- Why Use?
  - It may be too resource intensive (both time and space) to use a blind search
  - Even if a blind search will work we may want a more efficient search method

Heuristic Search Methods

- Methods that use a heuristic function to provide specific knowledge about the problem:
  - Heuristic Functions
  - Hill climbing
  - Greedy search
  - A* search algorithm

Heuristic Functions

- To further improve the quality of the previous methods, we need to include problem-specific knowledge on the problem
  - How can this be done in such a way that the algorithms remain generally applicable ???

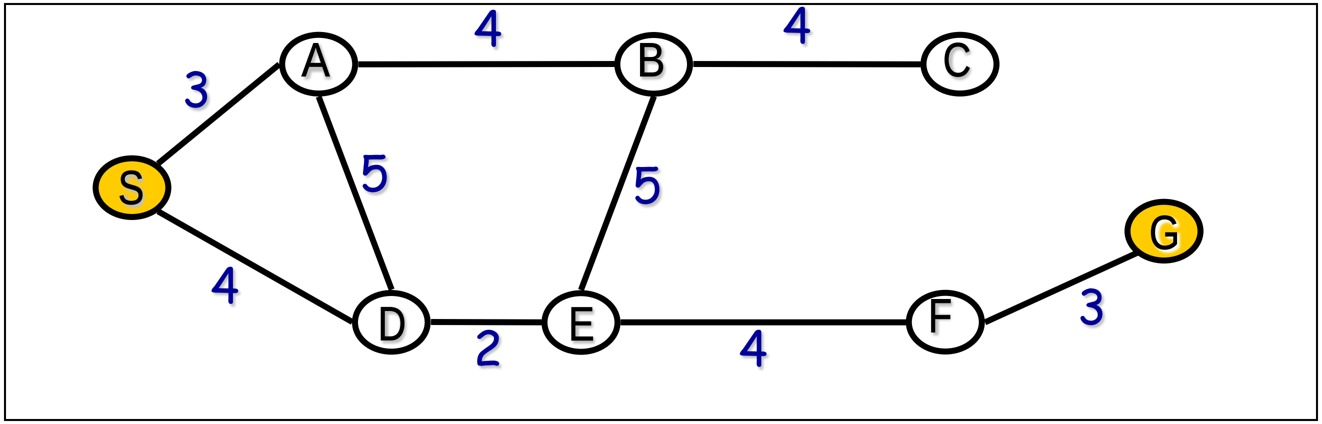
- HEURISTIC FUNCTIONS:
  - h: States $\rightarrow$ Numbers
  - h(n) : expresses the quality of the state n
    - allow to express problem-specific knowledge in the search method algorithm

Heuristic Functions

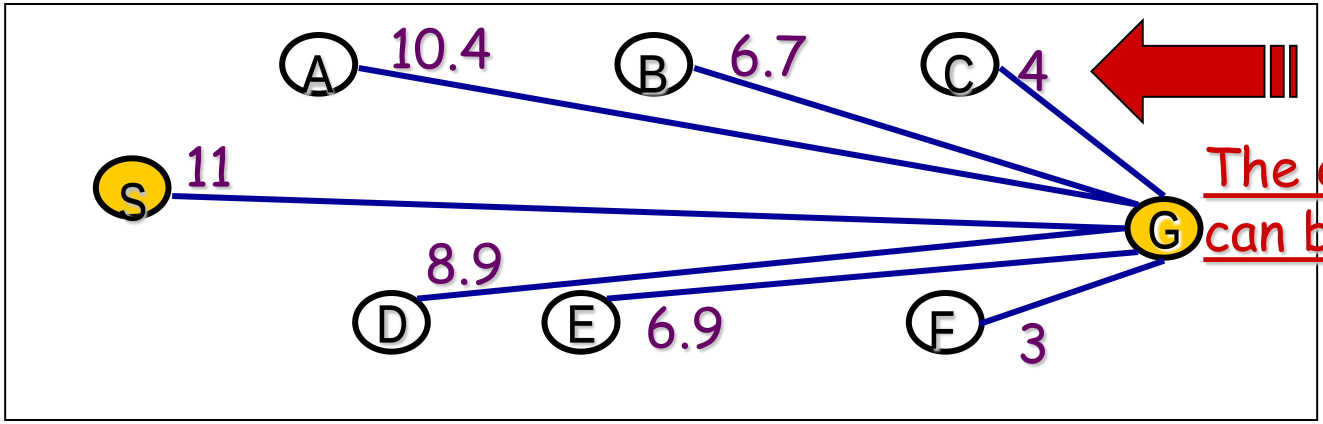
- **Heuristic function h(n),** takes a node n and returns a non-negative real number that is an estimate of the path cost from node n to a goal node
- The heuristic function is a way to inform the search about the direction to a goal
- It provides an informed way to guess which neighbor of a node will lead to a goal

Example 1: road map

- Imagine the problem of finding a route on a road map and that the NET below is the road map:



  - Define h(n) = the straight-line distance from n to G



The estimate can be wrong!

Example 2: 8-puzzle

- h1(n) = the number correctly placed tiles on the board:

$$h1 \begin{pmatrix} \boxed{1} & 3 & 2 \\ \boxed{8} & & \boxed{4} \\ 5 & \boxed{6} & 7 \end{pmatrix} = 4$$

- h2(n) = number or incorrectly placed tiles on board:
  - gives (rough!) estimate of how far we are from goal

$$h2 \begin{pmatrix} 1 & \boxed{3} & \boxed{2} \\ 8 & & 4 \\ \boxed{5} & 6 & \boxed{7} \end{pmatrix} = 4$$

Most often, 'distance to goal' heuristics are more useful !

Example 2: 8-puzzle Manhattan distance

- h3(n) = the sum of (the horizontal + vertical distance that each tile is away from its final destination):
  - gives a better estimate of distance from the goal node

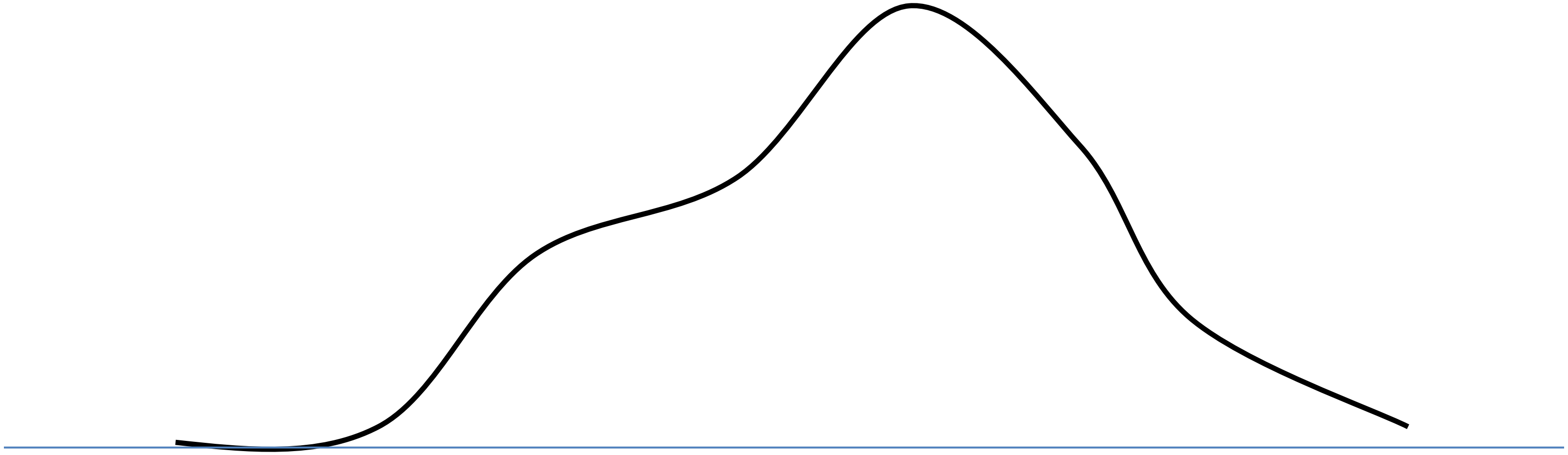
h3

1	3	2
8		4
5	6	7

= 1 + 1 + 2 + 2 = 6

Heuristic searches- Hill climbing

- A basic heuristic search method:
  - depth-first + heuristic

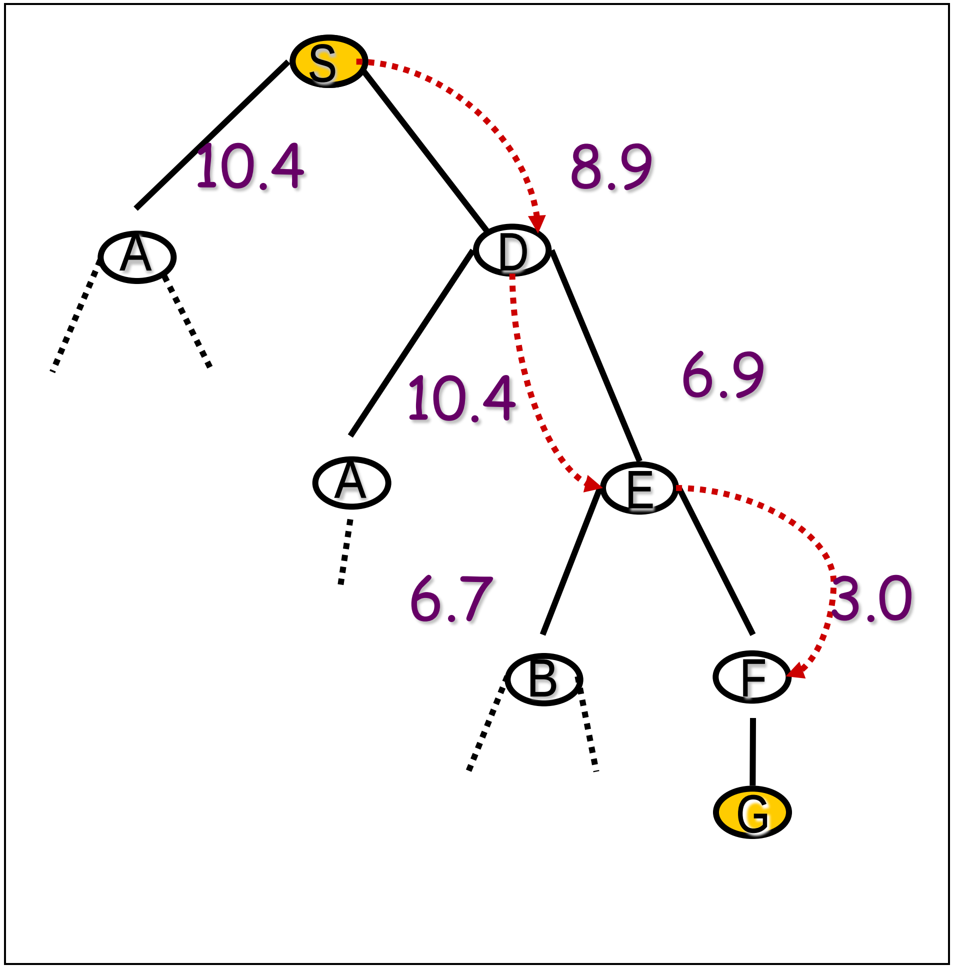


Hill climbing

- Described as: "Like climbing Everest in thick fog with amnesia"
- It is a mathematical optimization technique which belongs to the family local search
  - The most basic of all optimization techniques is evolution
- **Hill climbing** is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution
- If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found

Hill climbing_1

- Example: using the straight-line distance:

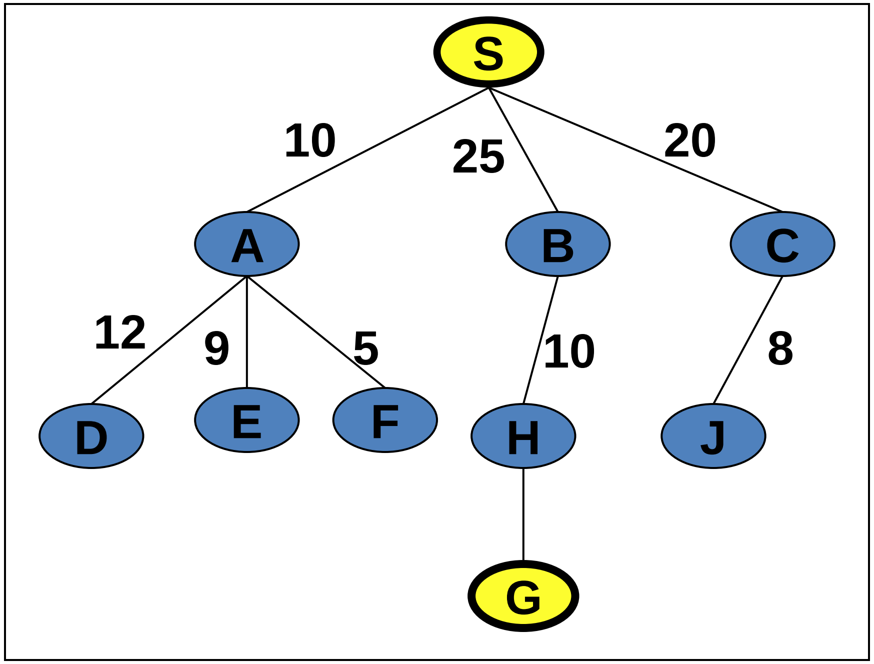


- Perform depth-first, <u>BUT</u>:
- instead of left-to-right selection,
- FIRST select the child with the <u>best heuristic</u> value

- **Note:** We are applying **minimal cost search** in this example

Minimal Cost Search Example

- Example: using the heuristic value



- If p is the current path:
  - the next path extends p by adding the node with <u>the smallest</u> cost from the endpoint of p

S

SA(10)

SAF(5)

...

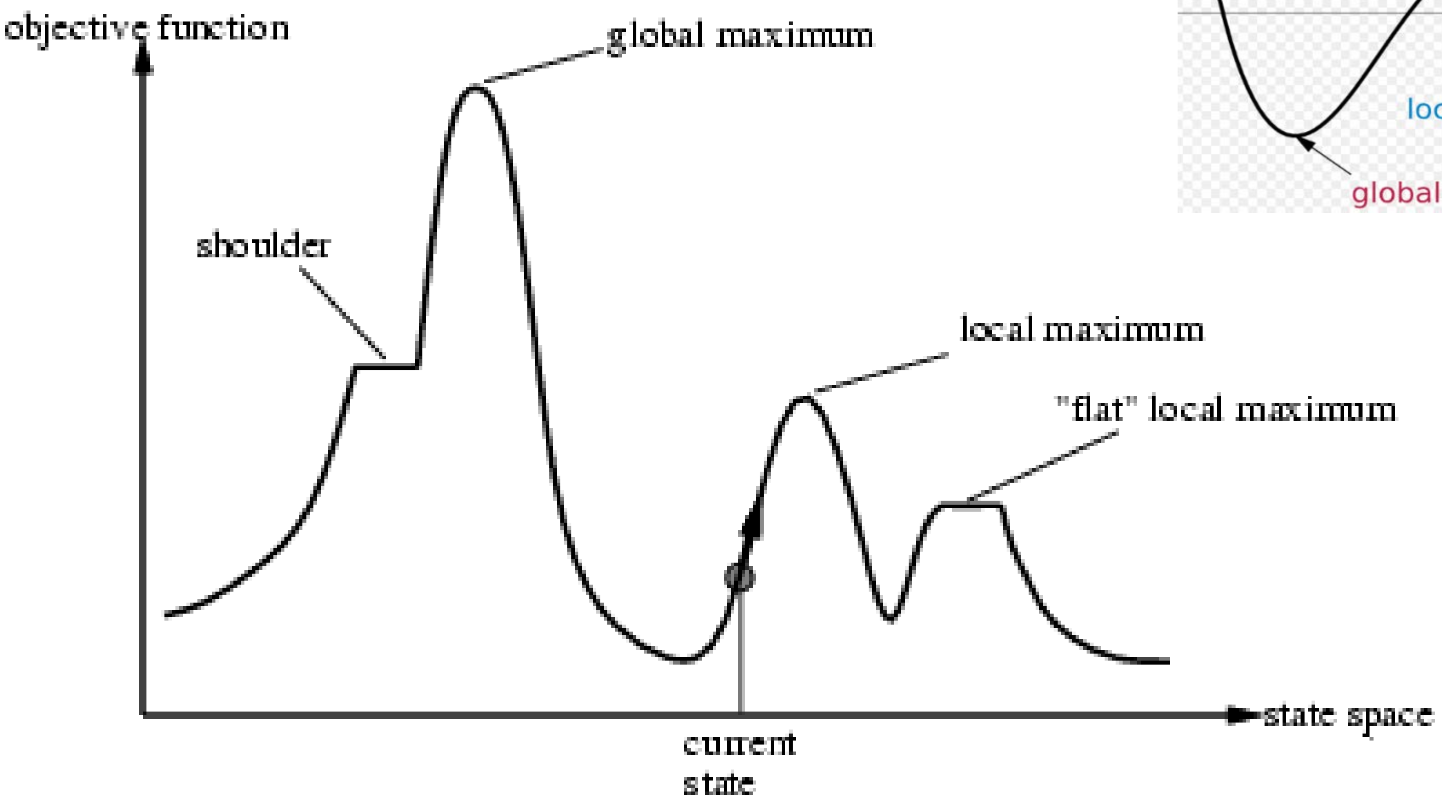
.....

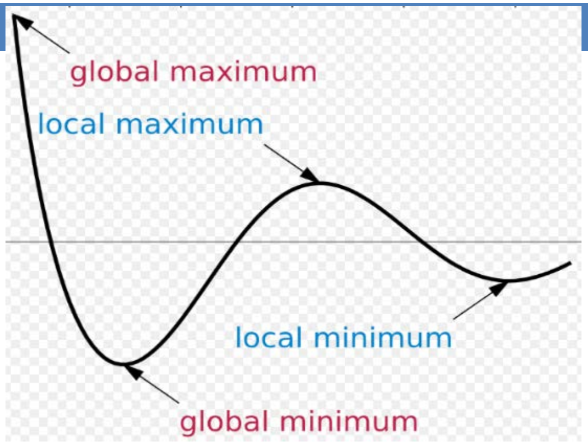
Hill climbing_2

- <u>Inspiring Example:</u> climbing a hill in the fog.
  - Heuristic function: check the change in altitude in 4 directions: the strongest increase is the direction in which to move next

- Is identical to Hill climbing_1, except for dropping the backtracking(loops)
- Produces a number of classical problems:
  - depending on initial state, can get stuck in local maxima

Problems with Hill climbing_2:



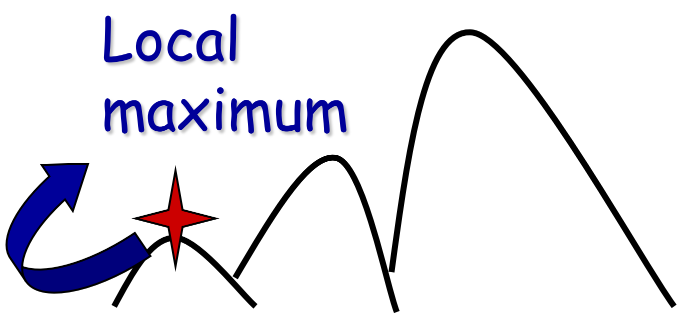


Problems with Hill climbing_2:

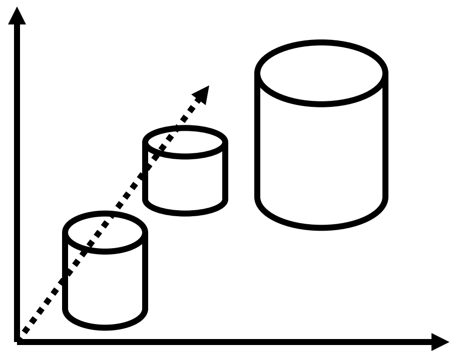
- Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :
  - **Local maximum :** At a local maximum all neighboring states have a values which is worse than the current state. Since hill climbing uses greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.
  - To overcome local maximum problem : Utilize backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.
  - **Plateau :** On plateau all neighbors have same value . Hence, it is not possible to select the best direction.
  - To overcome plateaus : Make a big jump. Randomly select a state far away from current state. Chances are that we will land at a non-plateau region
  - **Ridge :** Any point on a ridge can look like peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.
  - To overcome Ridge : In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

Problems with Hill climbing_2:

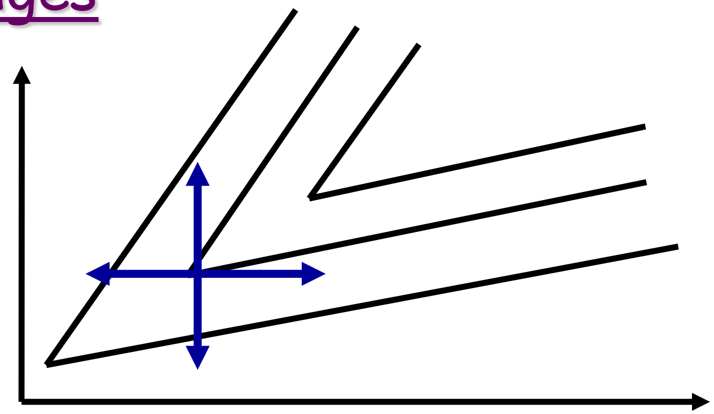
Foothills:



Plateaus



Ridges

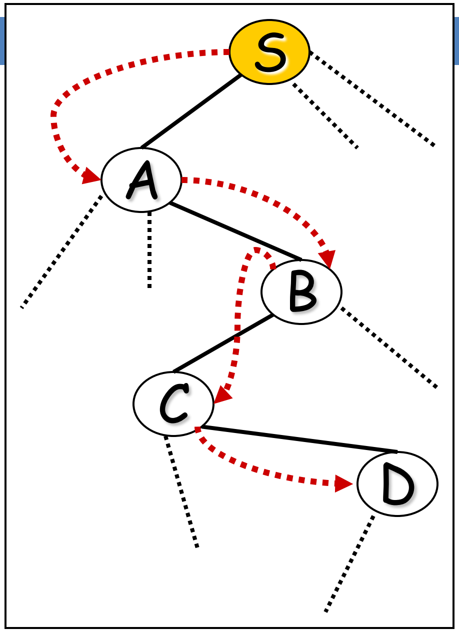


Comments:

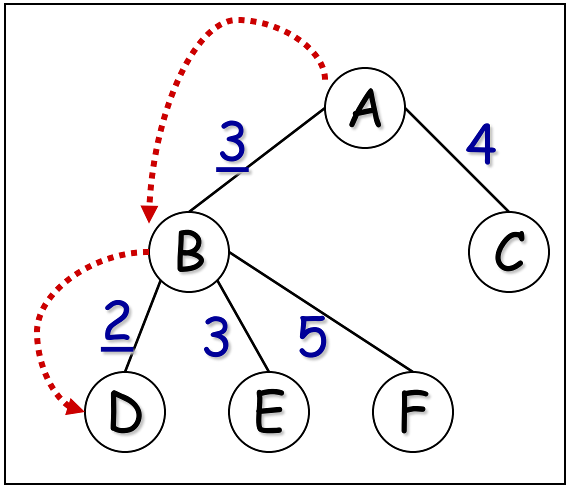
- Foothills are local maxima: it's a state that is better than all its neighbors but is not better than some other states farther away. hill climbing_2 can't detect the difference
- Plateaus is a flat area of the search space in which a whole set of neighboring states have the same value. It is not possible to determine the best direction in which to move and therefore doesn't allow you to progress in any direction
  - Foothills and plateaus require **random jumps** to be combined with the hill climbing algorithm
- Ridges neither: a ridge is a special kind of local maximum. It is an area of the search space that is higher than the surrounding areas and that itself has a slope. Any point on a ridge can look like a peak because the directions you have fixed in advance all move downwards for this surface
  - Ridges require **new rules**, more directly targeted to the goal, to be introduced (new directions to move)

Local search

- Hill climbing_2 is an example of local search.
- In local search, we only keep track of 1 path and use it to compute a new path in the next step.
  - QUEUE is always of the form:
    - (p)



- Another example:
  - MINIMAL COST search:
- If p is the current path:
  - the next path extends p by adding the node with the smallest cost from the endpoint of p



Heuristic Searches –
Best-First Search Algorithms

- Greedy Search algorithm
- A* Search algorithm

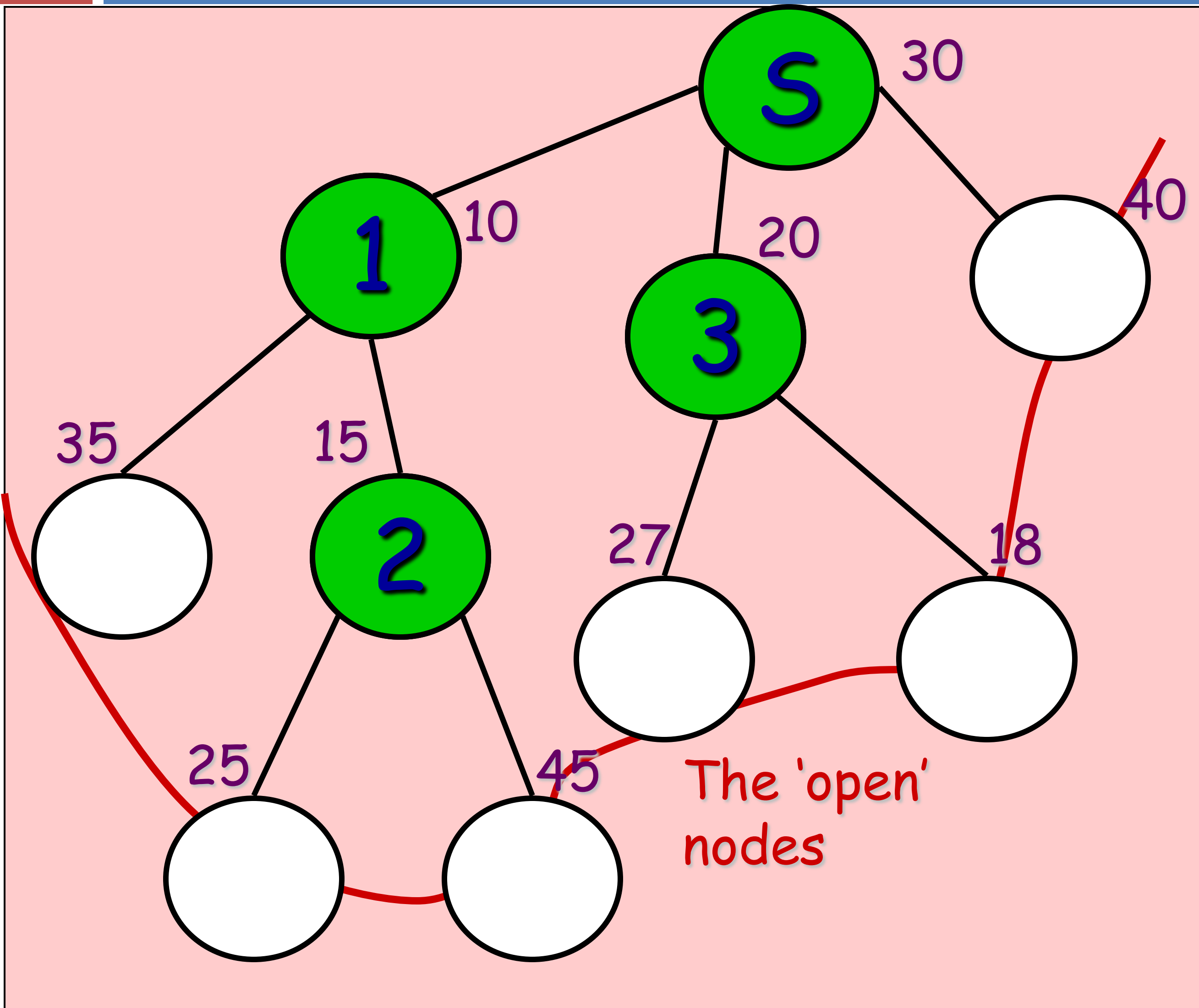
Greedy Search

- So named as it takes the biggest “bite” it can out of the problem
  - That is, it seeks to minimise the estimated cost to the goal by expanding the node estimated to be closest to the goal state

in other words,

- Always expand the heuristically best nodes first

Greedy Search, or
Heuristic best-first search:



- At each step, select the node with the best (in this case: <u>lowest</u>) heuristic value

Greedy Search algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
AND goal is not reached

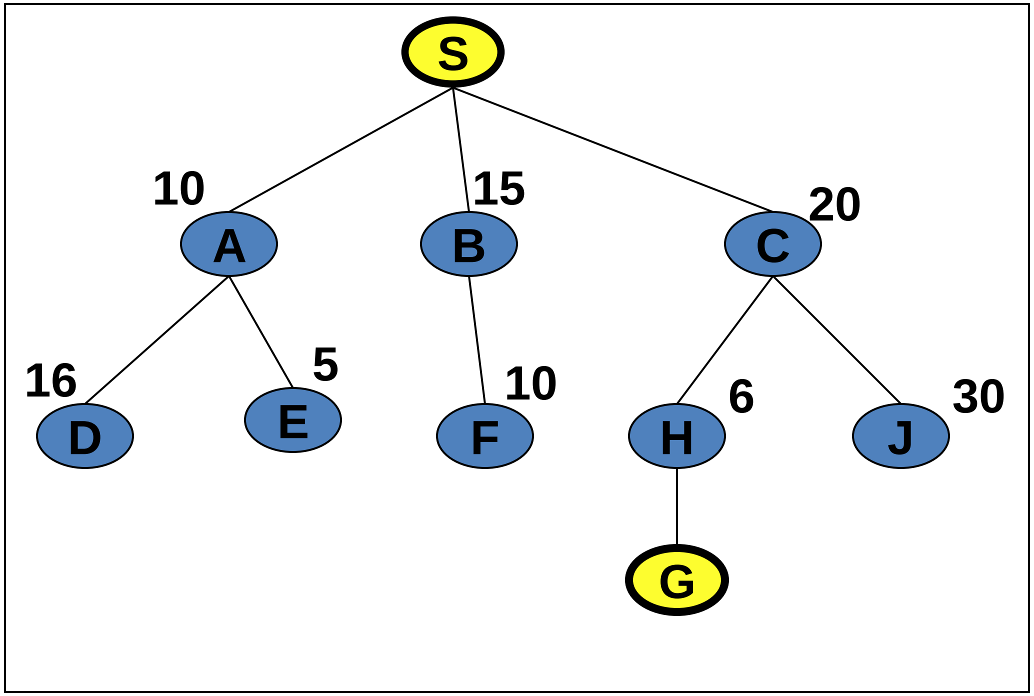
DO remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
add the new paths and sort the entire QUEUE;

(HEURISTIC)

3. IF goal reached
THEN success;
ELSE failure;

Greedy Search example

- Example: using the heuristic value



- At each step, select the node with the best (in this case: <u>lowest</u>) heuristic value

S

SA(10),SB(15),SC(20)

SAD(16),SAE(5),SB(15),SC(20)

SAE(5),SB(15),SAD(16),SC(20)

SAE(5),SB(15),SAD(16),SC(20)

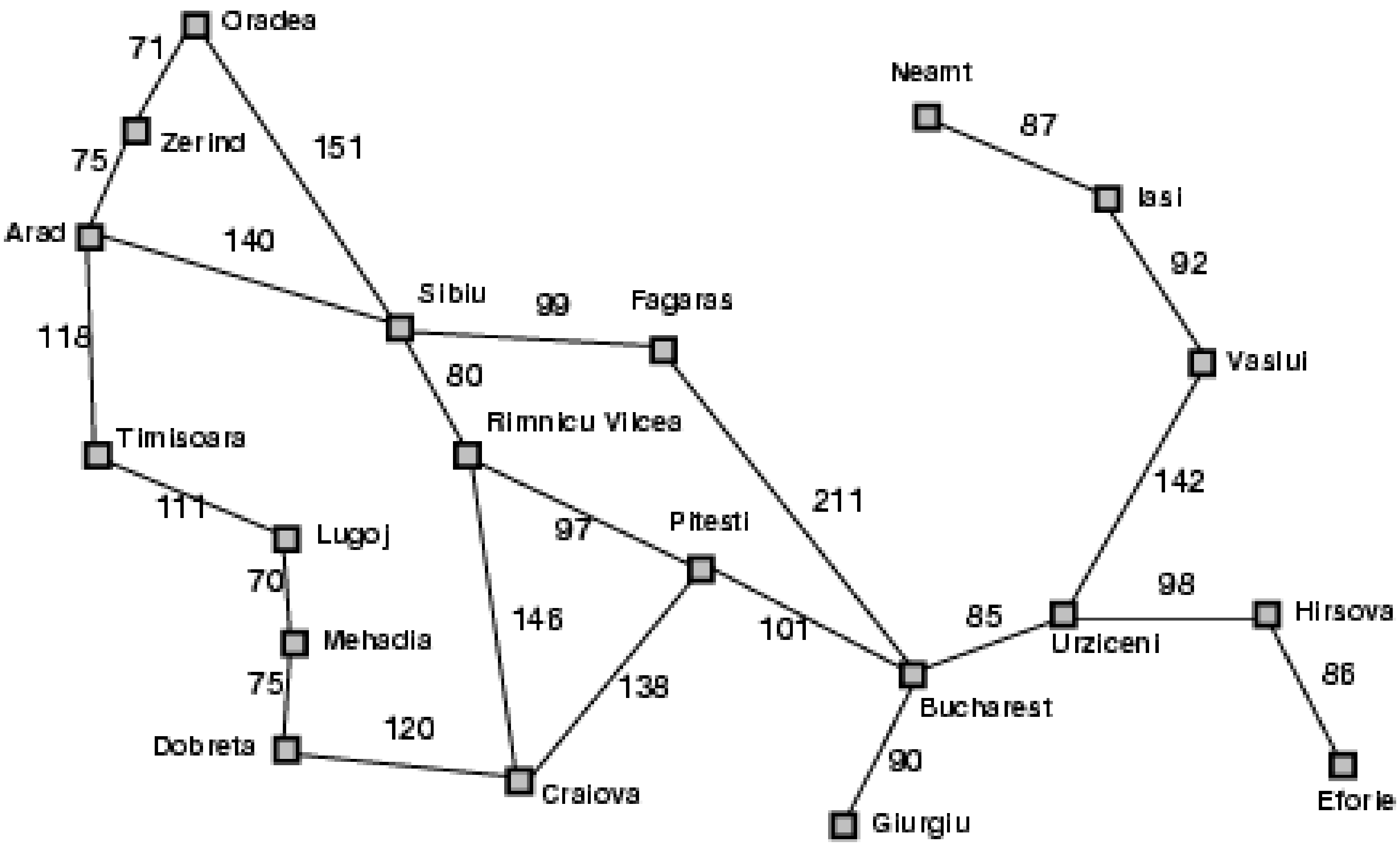
SBF(10),SAD(16),SC(20)

SBF(10),SAD(16),SC(20)

SCH(6),SCJ(30)

SCHG <=goal

Example: Greedy algorithm for Romania Tour with step costs in km



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example cont…

- Evaluation function $f(n) = h(n)$ (heuristic)

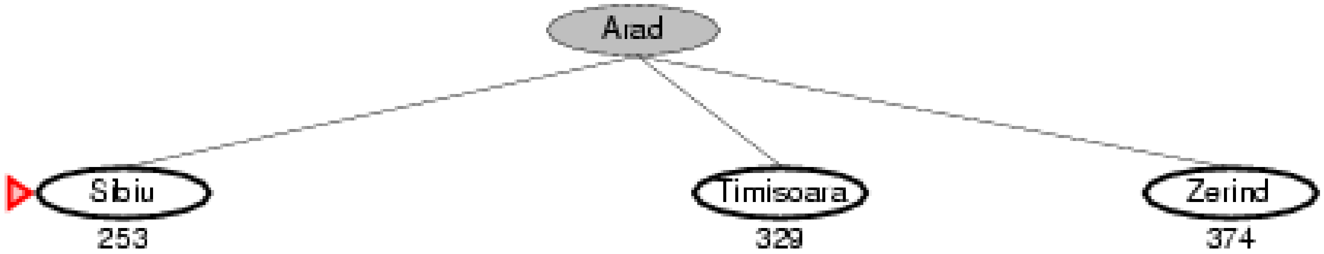
  = estimate of cost from n to goal

- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

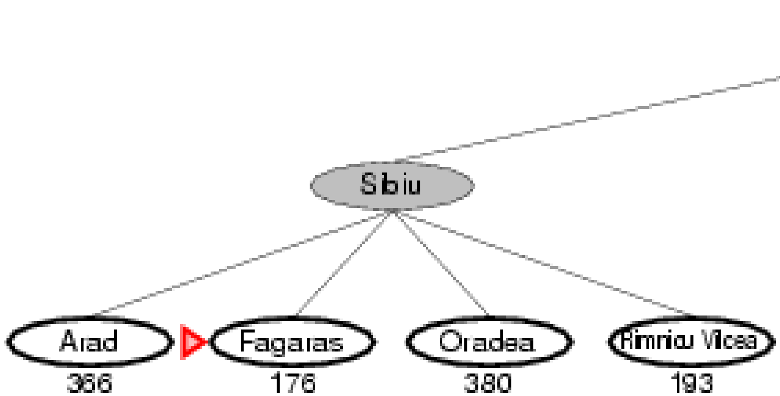
- Greedy best-first search expands the node that appears to be closest to goal

Example cont…

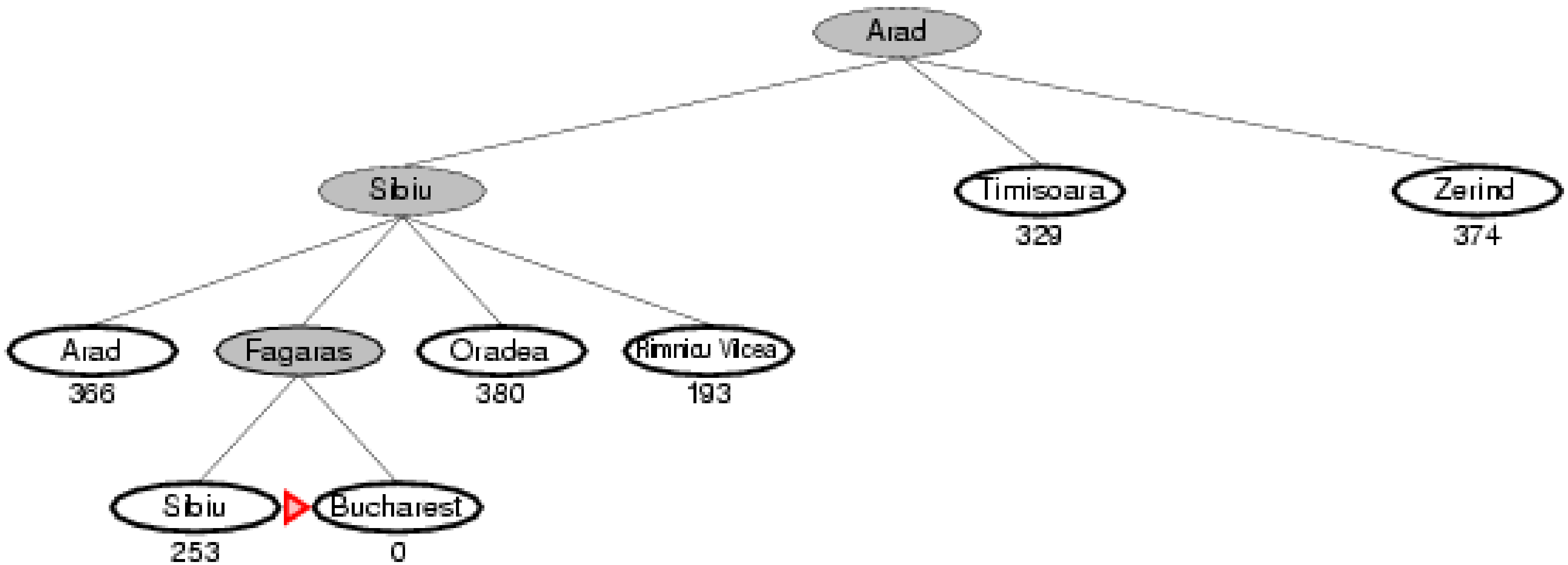
Example cont…



Example cont…



Example cont…



Heuristic Searches - Greedy Search

- It is only concerned with short term aims
- It is not optimal
- It is not complete
  - It is possible to get stuck in an infinite loop, unless you check for repeated states
  - e.g., Iasi → Neamt → Iasi → Neamt →
- Time complexity $O(b^m)$, but a good heuristic can give dramatic improvement
- Space complexity $O(b^m)$, keeps all nodes in memory

Heuristic Searches - A* Algorithm

- A combination of Greedy search and Uniform cost search.
  - to compliment one another
- This search method minimises the cost to the goal using an heuristic function, h(n)
  - Greedy search can considerably cut the search time but it is neither optimal nor complete
- By comparison uniform cost search minimises the cost of the path so far, g(n)
  - Uniform cost search is both optimal and complete but can be very inefficient

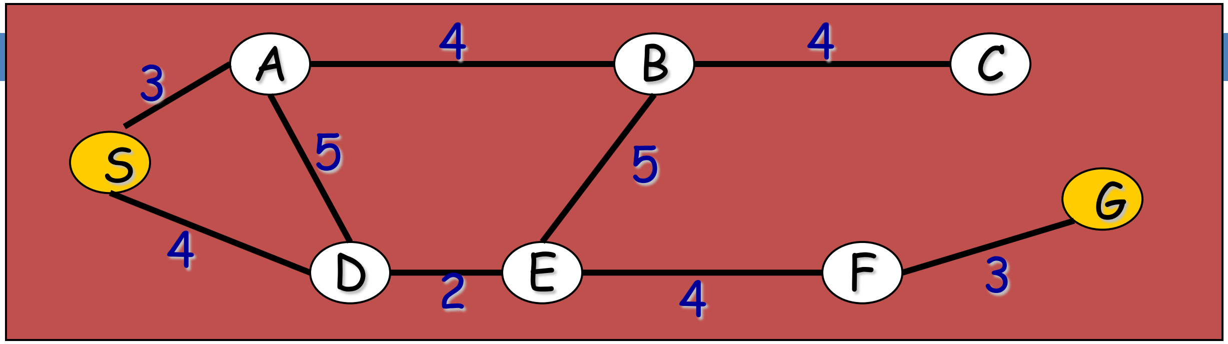
Heuristic Searches - A* Algorithm

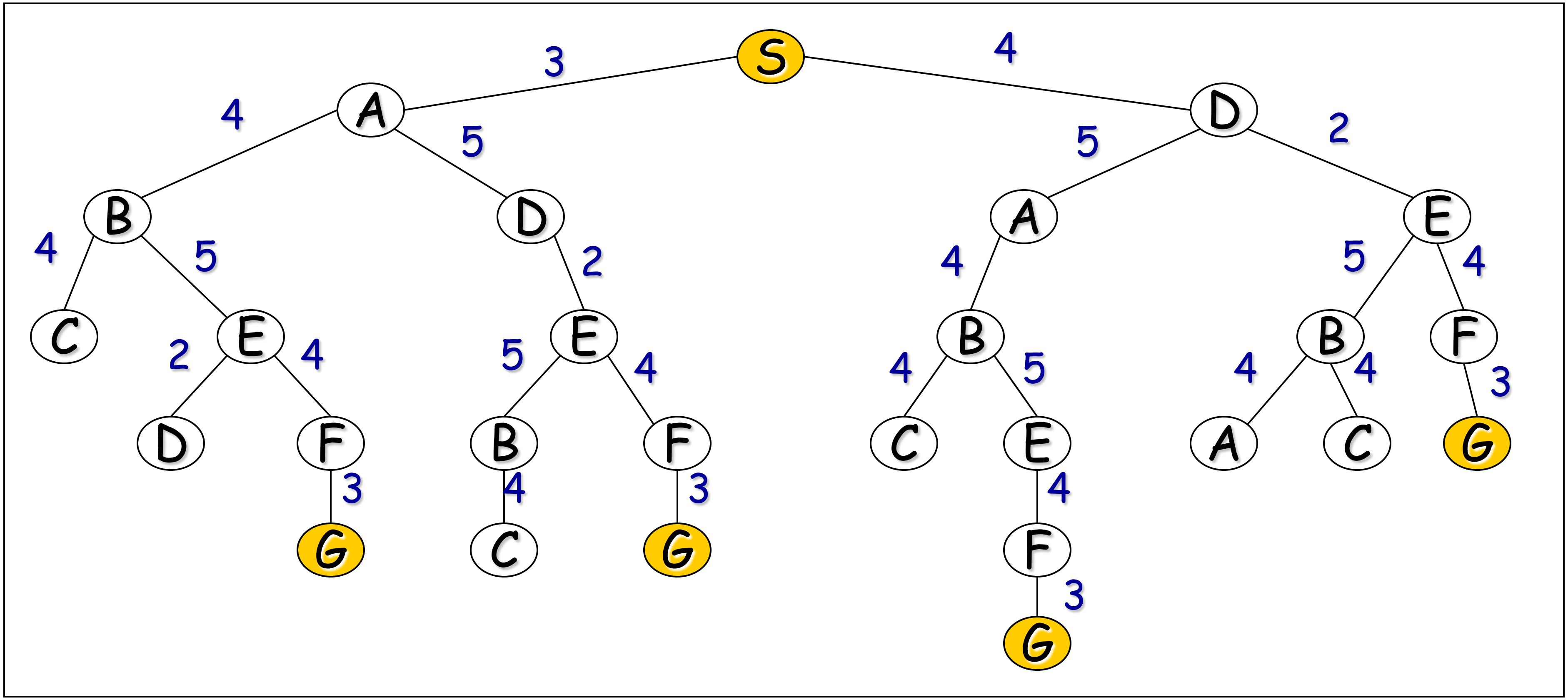
- Idea: avoid expanding paths that are already expensive
- Always expand the path that has a minimum value of $f(n)$

- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated cost from n to goal
  - $f(n)$ = estimated total cost of path through n to goal

Road map example:

Re-introduce the costs of paths in the NET

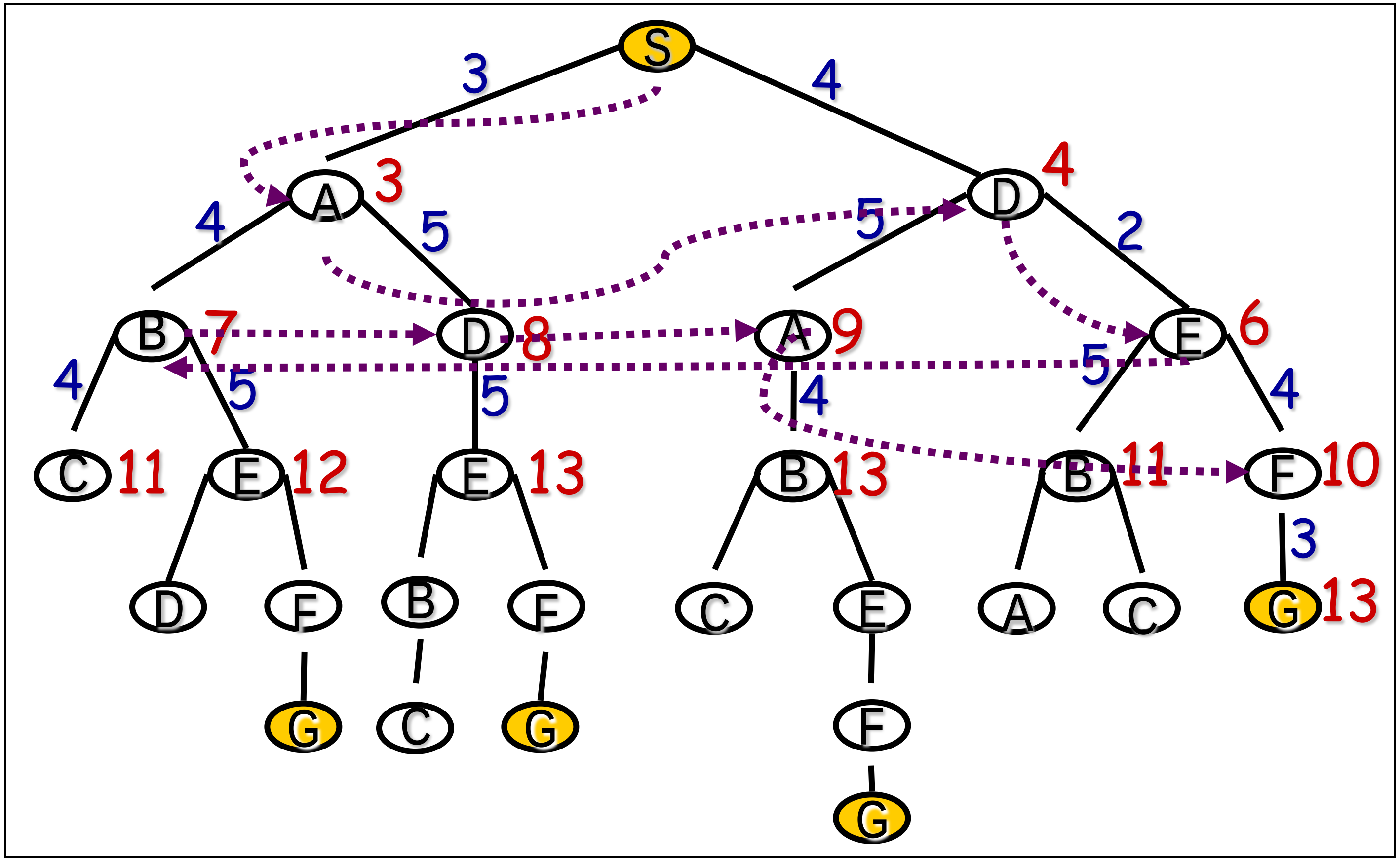




A look at Uniform cost search = uniformed best-first

- The Uniform Cost Search is also referred to as Cheapest-First Search best because it is guaranteed to find the path with the cheapest total cost.
- It is also referred to as **Dijkstra's** single-source shortest algorithm
- The Uniform Cost Search is related to the Breadth-First Search
- It determines the node to be expanded next by calculating the cost so far to reach each node in the frontier from the root and picks the path with the lowest total cost

A look at Uniform cost search
= uniformed best-first



- At each step, select the node with the lowest accumulated cost.

- with g(n) = the sum of edge costs from start to n

Now incorporate heuristic estimates

- Replace the 'accumulated cost' in the 'Uniform cost search' by a function:

$$\boxed{\underbrace{f(path)}_{F(n)} = \underbrace{cost(path)}_{g(n)} + \underbrace{h(endpoint\_path)}_{h(n)}}$$

- where:

cost(path) = the accumulated cost of the partial path
h(n) = a heuristic estimate of the cost remaining from n to a goal node

f(path) = an estimate of the cost of a path extending the current path to reach a goal

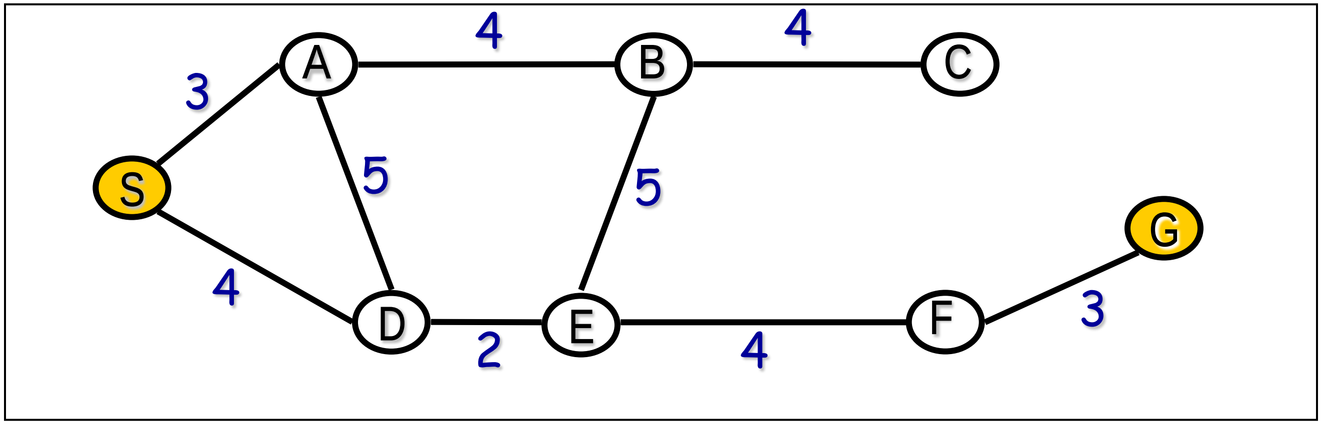
$$\boxed{F(n) = g(n) + h(n)}$$

Heuristic Searches - A* Algorithm

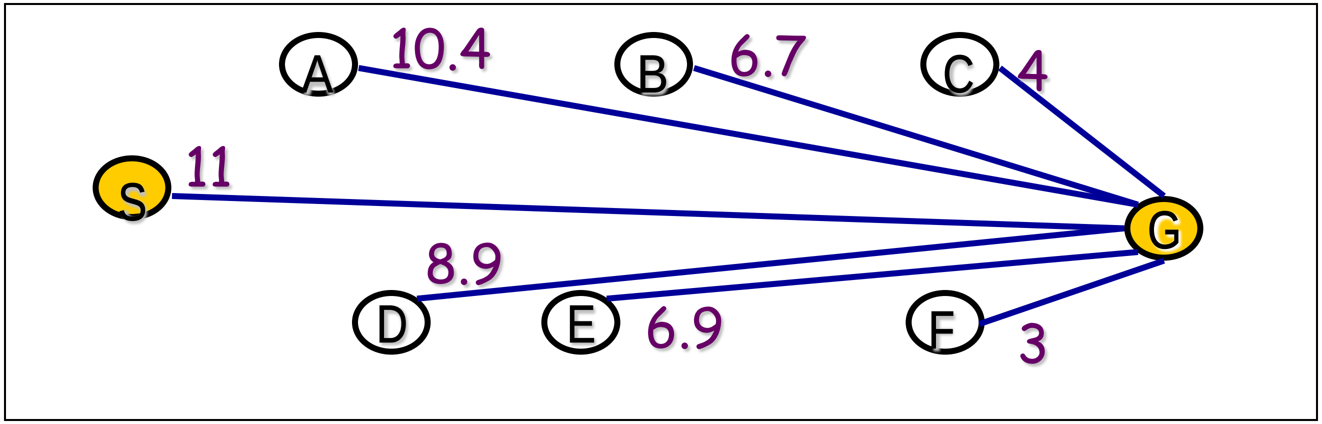
- Combines the cost so far and the heuristic estimated cost to the goal. That is $f(n) = g(n) + h(n)$
  This gives us estimated cost of the cheapest solution through path *n*
- It can be proved to be **optimal** and **complete** providing that the heuristic is **admissible**
  - That is the heuristic must never over estimate the cost to reach the goal
- But, the number of nodes that have to be searched still grows exponentially

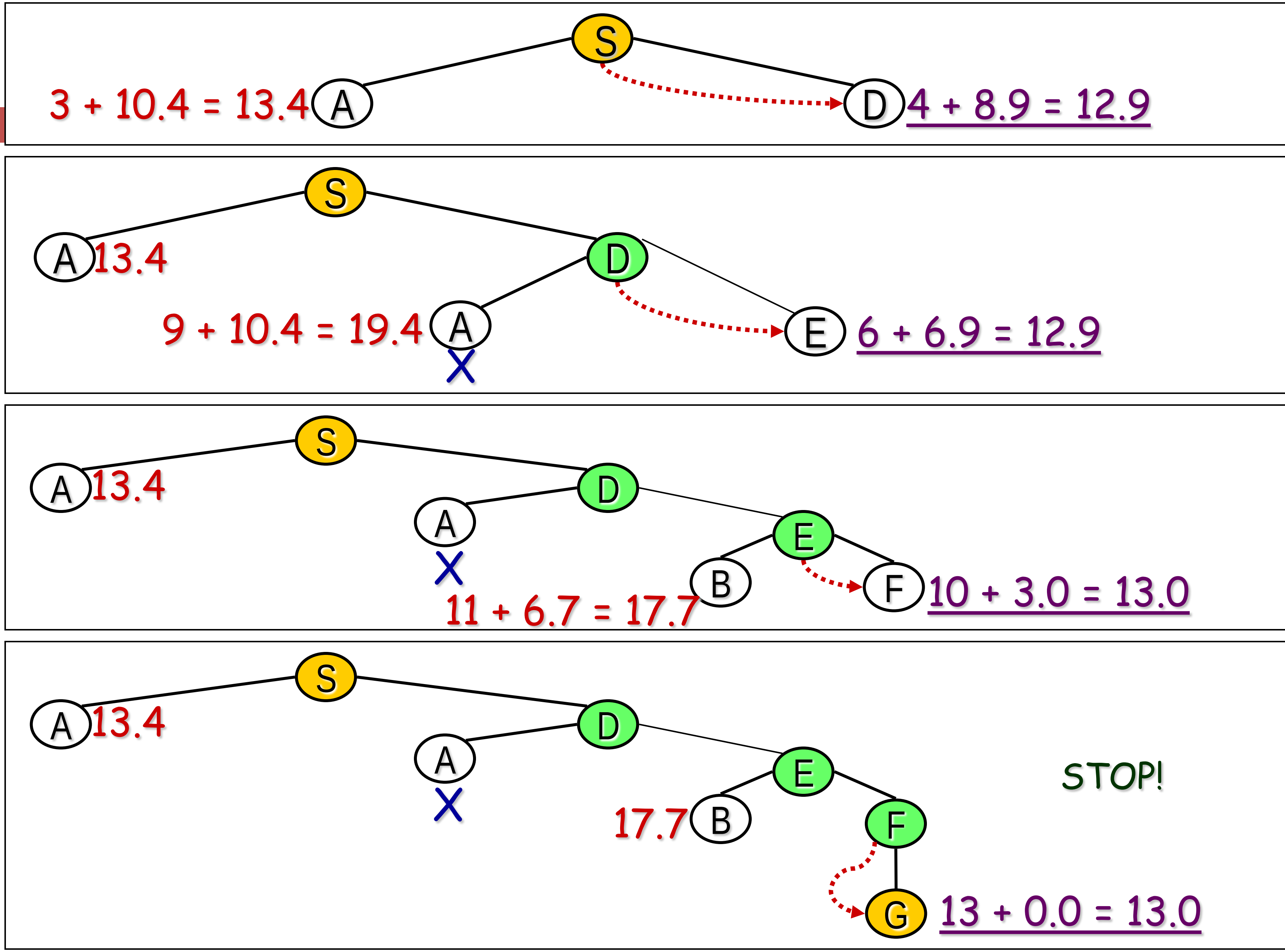
Example: road map

- Imagine the problem of finding a route on a road map. The paths distances between nodes define g(n):



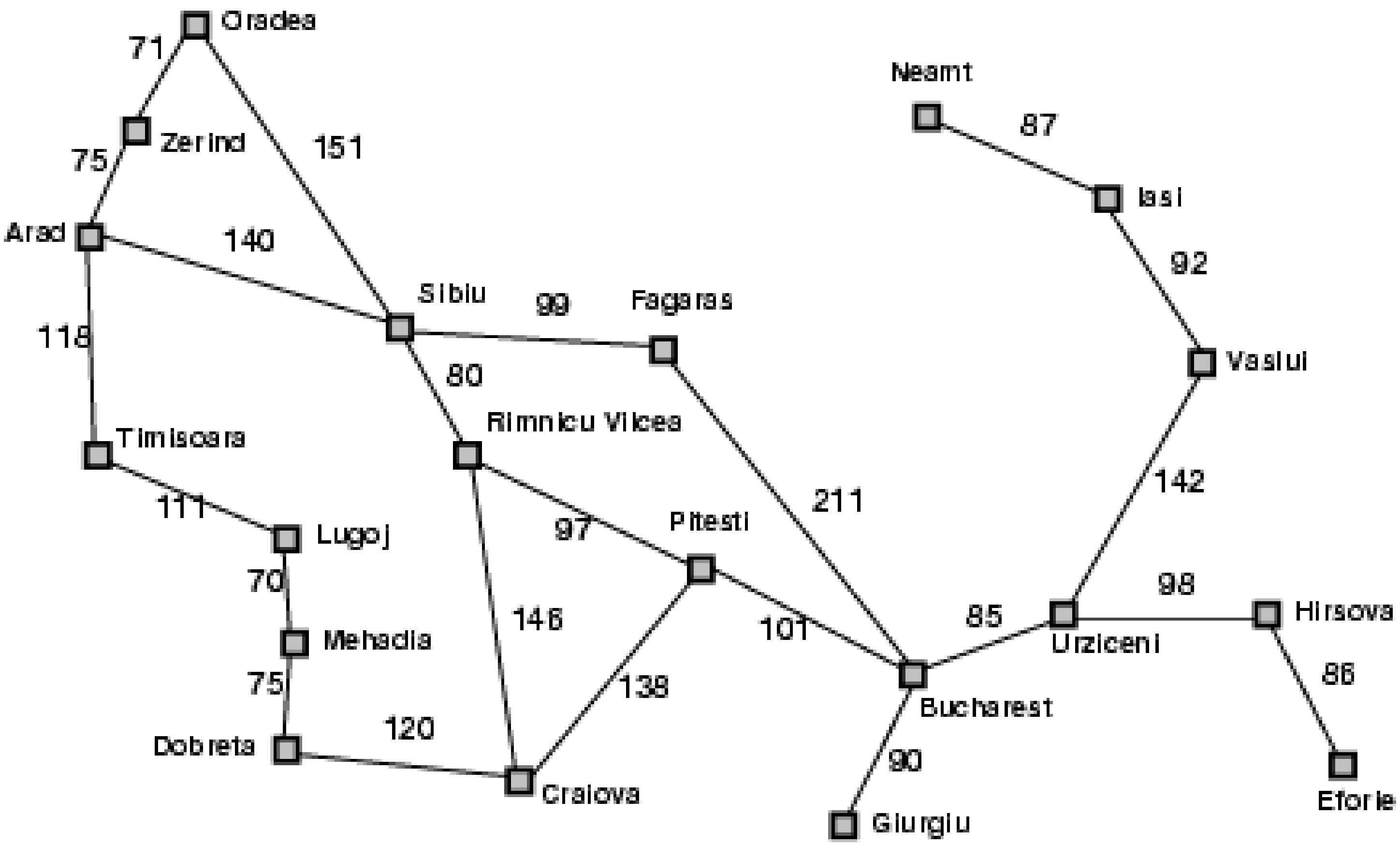
- Define h(n) = the straight-line distance from node to G





S
3 + 10.4 = 13.4
A
D
4 + 8.9 = 12.9
S
A
13.4
D
9 + 10.4 = 19.4
A
E
6 + 6.9 = 12.9
S
A
13.4
D
A
E
B
F
10 + 3.0 = 13.0
11 + 6.7 = 17.7
S
A
13.4
D
A
E
STOP!
17.7
B
F
G
13 + 0.0 = 13.0

Example: A* algorithm for Romania Tour with step costs in km

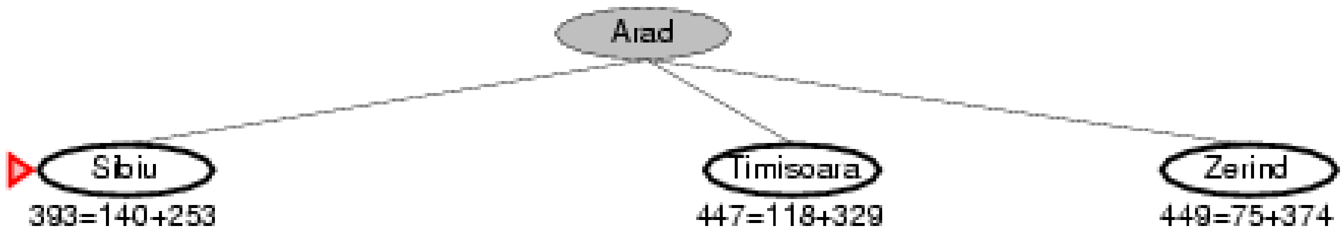


Straight–line distance to Bucharest

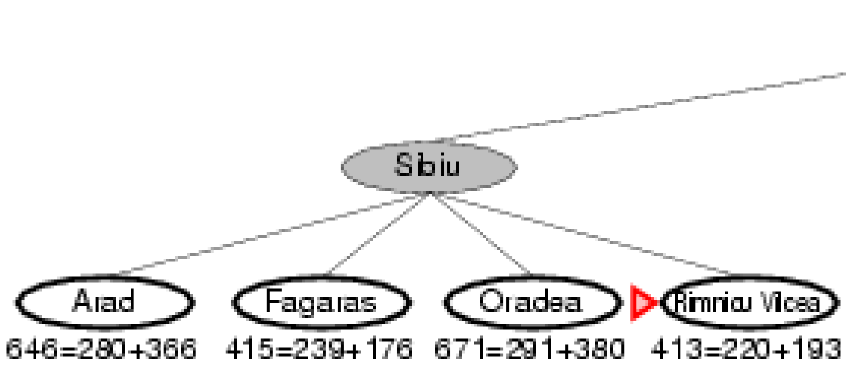
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example:
A* Algorithm for Romania Tour

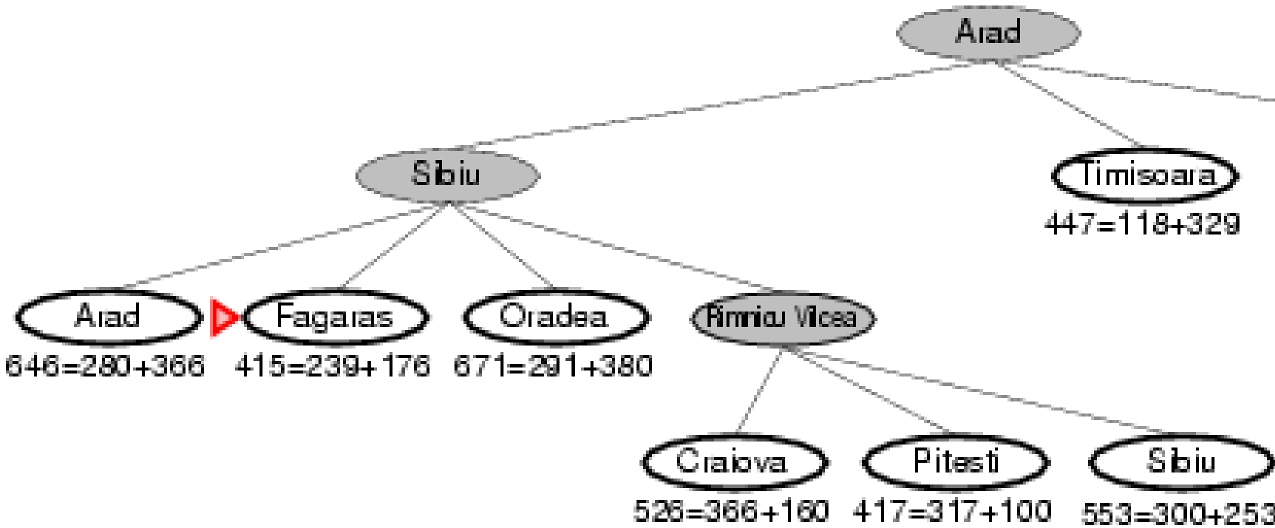
Example cont…



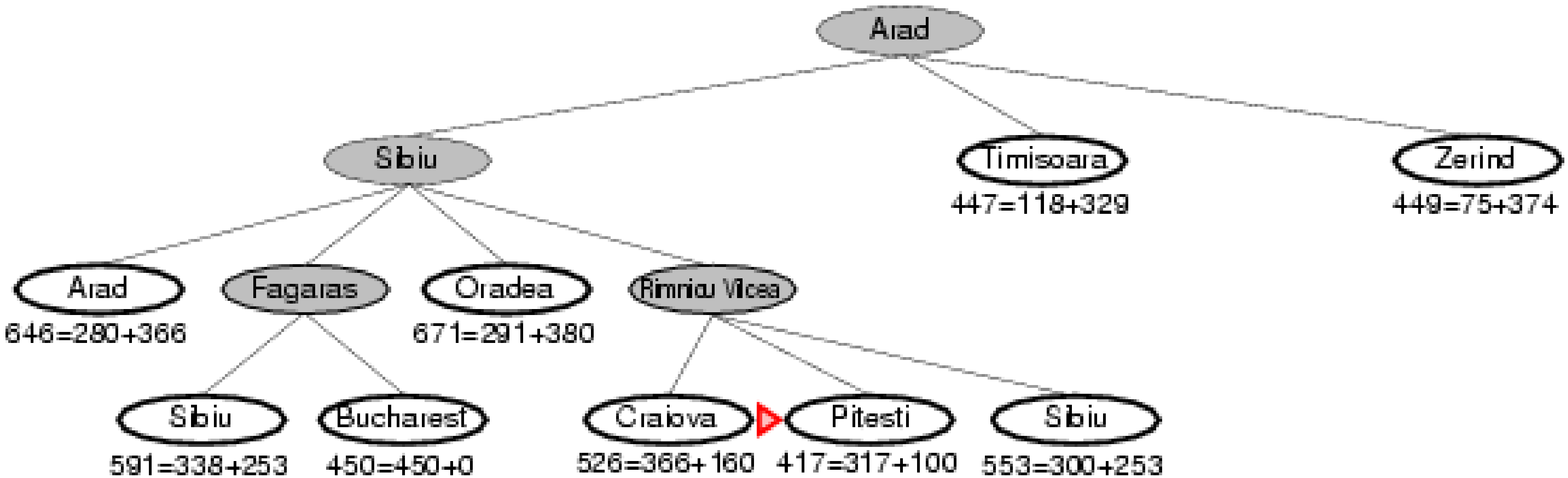
Example cont...



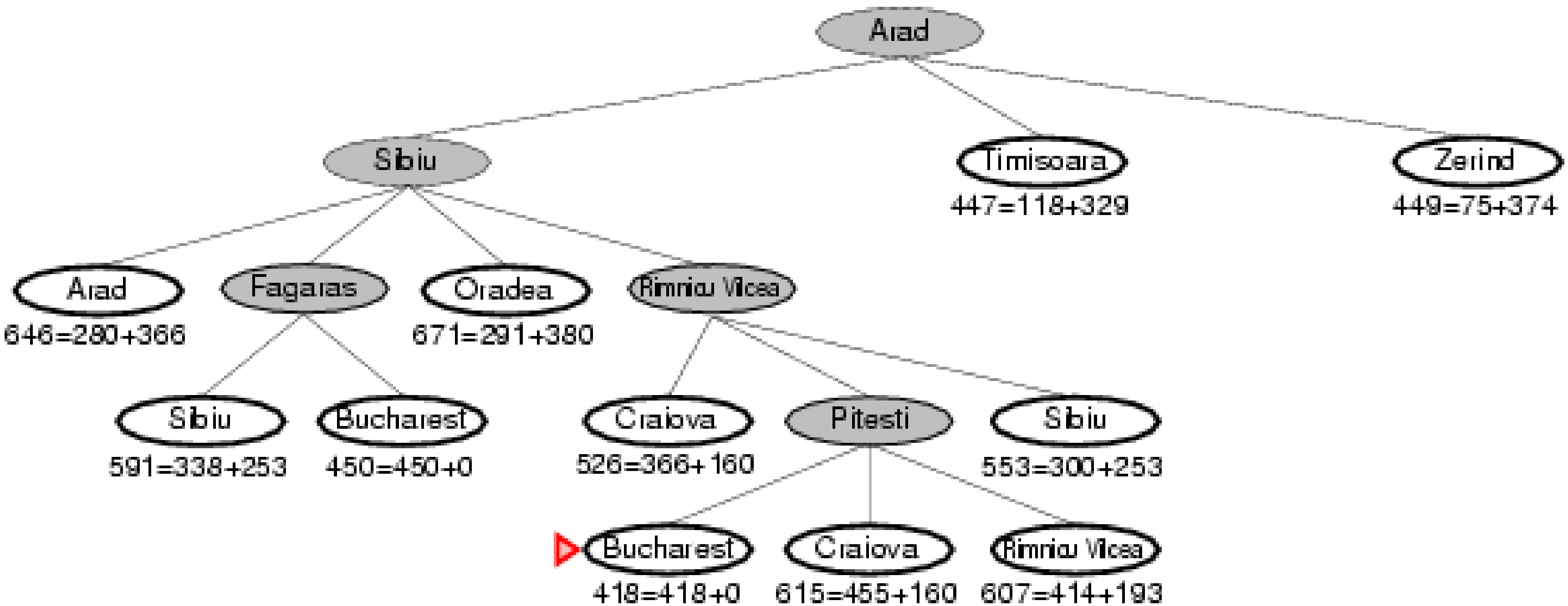
Example cont…



Example cont…



Example cont…



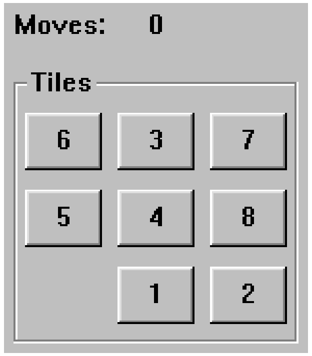
Example: A* Algorithm for 8-Puzzle

5	4	
6	1	8
7	3	2

Initial State

1	2	3
8		4
7	6	5

Goal State



- Typical solution is about twenty steps
- Branching factor is approximately three. Therefore a complete search would need to search 3^{20} states. But by keeping track of repeated states we would only need to search 9! (362,880) states
- But even this is a lot (imagine having all these in memory)
- Our aim is to develop a heuristic that does not over estimate (it is admissible) so that we can use A* to find the optimal solution

Heuristic Searches - Possible Heuristics

- H_1 = the number of tiles that are in the wrong position (=7)

- H_2 = the sum of the distances of the tiles from their goal positions using the Manhattan Distance (=18)

Both are admissible but which one is best?

Test from 100 runs with varying solution depths

Number of nodes expanded

	Search Cost		
Depth	**IDS**	**A*(h_1)**	**A*(h_2)**
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	364404	227	73
14	3473941	539	113
16		1301	211
18		3056	363
20		7276	676
22		18094	1219
24		39135	1641

H_2 looks better as fewer nodes are expanded. But why?

Effective Branching Factor

	Search Cost			EBF		
Depth	**IDS**	**A*(h_1)**	**A*(h_2)**	**IDS**	**A*(h_1)**	**A*(h_2)**
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23

- H_2 has a lower branching factor and so fewer nodes are expanded
- Therefore, one way to measure the quality of a heuristic is to find its average branching factor
- H_2 has a lower EBF and is therefore the better heuristic

Summary

- Blind Search is very expensive
- A* Search with info (heuristics) is far better
- Info can be difficult to incorporate
- The more info, the better